Compiler Techniques for High Performance Computing, Energy Efficiency, and Resilience

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Sanket Tavarageri, M.S.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2014

Dissertation Committee:

P. Sadayappan, Advisor
Atanas Rountev
Paul A. G. Sivilotti
Abstract

The technology trends have brought parallel computers — multi-core and many-core processors to the mainstream. Parallel computing presents formidable challenges for compilers from performance and energy view-points: how to generate codes that achieve portable high performance on different processor architectures; how to support cache coherence for parallel processors at a reasonable hardware and energy cost etc. This dissertation develops solutions to both of the aforementioned questions — we propose novel adaptive tiling algorithms that enable dynamic change of tile sizes so that software can be adapted to different execution environments by altering tile sizes. Compiler techniques are developed for software cache coherence in order that expensive hardware cache coherence methods can be replaced or simplified to reduce energy and hardware expenditure associated with the task of providing cache coherence.

The technology scaling has improved processor performance at a much faster rate than memory performance. As a result, current processor architectures incorporate large cache memories in an attempt to bridge the gap between computing and memory-access speeds (cache memories can fetch reused data much faster compared to main memory). This dissertation presents compiler techniques to tackle leakage (static) energy problem in cache memories. Compiler algorithms presented analyze data reuse pattern in application codes and derive the amount of cache that has to be kept on to not affect performance. The excessive cache capacity can be turned off to save leakage energy.
The smaller technology feature sizes make hardware susceptible to transient faults. The implication for memory subsystem (including cache memories, main memory and data-path) is that high-energy particle strikes may cause bit-flips and hence, affect correctness of program results. In this dissertation, a compiler approach to introduce checksum computations in the application program is developed to detect memory errors. The techniques are light-weight in terms of performance and memory bandwidth requirements.

In sum, the dissertation makes contributions to advance compiler technology to achieve high-performance for parallel applications, to reduce energy costs, and to effectively address transient hardware errors.
Acknowledgments

Education is a journey and so is research; a Ph.D is an important milestone in that journey. I am glad and grateful that I have been able to reach this stage and my thanks are due to everyone who has enabled and propelled me to walk on the road of education and research.

First and foremost, I would like to sincerely thank my advisor, Saday. From the beginning of grad-school, Saday has been a constant encouragement and a tireless mentor. He brings passion and genuine curiosity to work. His insights and in-depth comments on the subject matter helped me chart my research course and I will forever be grateful for his advisement and equally for his benevolent disposition.

I also profusely thank Nasko and Paul for their guidance and for serving on my dissertation committee. I would like to gratefully acknowledge fruitful research collaborations with Sriram Krishnamoorthy, Louis-Noël Pouchet, Ram Ramanujam, Wooil Kim, Josep Torrellas, Muthu Baskaran, and Albert Hartono.

Words cannot adequately express my love and affection for my parents, wife Aditi, and the rest of my family; their support throughout has been crucial in bringing this dissertation to life. I also have been fortunate to have had the chance to interact with many interesting and intellectual grad-school friends – it was fun.
Vita

May 2007 ................................. Bachelor of Technology,
Computer Science & Engineering,
National Institute of Technology Kar-
nataka, Surathkal, India

July 2007 — July 2009 .................. Software Development Engineer,
Microsoft, India

September 2009 — Present ............. Graduate Research Associate,
The Ohio State University

June 2012 ................................. M.S.,
The Ohio State University

August 2012 — May 2013 ............... Graduate Teaching Associate,
The Ohio State University

Publications

Research Publications

Sanket Tavarageri, Sriram Krishnamoorthy, P. Sadayappan.
Compiler-Assisted Detection of Transient Memory Errors
In the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI), June 2014

Sanket Tavarageri, Wooil Kim, Josep Torrellas, P. Sadayappan.
Automatic Generation of Coherence Instructions for Software-Managed Multiprocessor Caches
Technical report OSU-CISRC-1/14-TR02, 2014

Sanket Tavarageri, J. Ramanujam, P. Sadayappan.
Adaptive Parallel Tiled Code Generation and Accelerated Auto-tuning
In the International Journal of High Performance Computing Applications (IJHPCA), 2013
Sanket Tavarageri, P. Sadayappan.
**A Compiler Analysis to Determine Useful Cache Size for Energy Efficiency**
In the Workshop on High-Performance, Power-Aware Computing (HPPAC), May 2013

**Dynamic Selection of Tile Sizes**
In the 18th Annual International Conference on High Performance Computing (HiPC), December 2011

**Parametric Tiling of Affine Loop Nests**
In the Workshop on Compilers for Parallel Computing (CPC), July 2010

Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Tom Henretty, J. Ramanujam, P. Sadayappan.
**Parameterized Tiling Revisited**
In the International Symposium on Code Generation and Optimization (CGO), April 2010

**Fields of Study**

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Adaptive Parallel Tiled Code Generation and Accelerated Auto-tuning</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Background</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Parametric Tiling</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Parallel Parametric Tiling</td>
<td>10</td>
</tr>
<tr>
<td>2.3 Tile Size Adaptation for Sequential Tiling</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Tile Size Adaptation for Outer-loop Parallel Tiling</td>
<td>14</td>
</tr>
<tr>
<td>2.5 Tile size Adaptation for Wavefront Parallel Tiling</td>
<td>16</td>
</tr>
<tr>
<td>2.5.1 Tile Size Adaptation Challenges</td>
<td>17</td>
</tr>
<tr>
<td>2.5.2 Overview of the Solution</td>
<td>18</td>
</tr>
<tr>
<td>2.5.3 Code Generation</td>
<td>20</td>
</tr>
<tr>
<td>2.6 Experimental Evaluation</td>
<td>24</td>
</tr>
<tr>
<td>2.6.1 Benchmarks and Set-up</td>
<td>24</td>
</tr>
<tr>
<td>2.6.2 Methodology</td>
<td>25</td>
</tr>
<tr>
<td>2.6.3 Results</td>
<td>28</td>
</tr>
<tr>
<td>2.7 Related Work</td>
<td>33</td>
</tr>
<tr>
<td>2.8 Conclusion</td>
<td>34</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A 2-D loop-nest</td>
</tr>
<tr>
<td>2.2</td>
<td>Sequential Tiled Code</td>
</tr>
<tr>
<td>2.3</td>
<td>(Optimized) Sequential Tiled Code</td>
</tr>
<tr>
<td>2.4</td>
<td>Adaptive Sequential Tiled Code</td>
</tr>
<tr>
<td>2.5</td>
<td>Original Tiled Space</td>
</tr>
<tr>
<td>2.6</td>
<td>New Tiled Space</td>
</tr>
<tr>
<td>2.7</td>
<td>Outer-loop Parallel Tiled Code</td>
</tr>
<tr>
<td>2.8</td>
<td>Adaptive Outer-loop Parallel Tiled Code</td>
</tr>
<tr>
<td>2.9</td>
<td>Wavefront Parallel Tiled Code</td>
</tr>
<tr>
<td>2.10</td>
<td>Adaptive Wavefront Parallel Tiled Code</td>
</tr>
<tr>
<td>2.11</td>
<td>Wavefronts in Original Tiled Space</td>
</tr>
<tr>
<td>2.12</td>
<td>Wavefronts in New Tiled Space</td>
</tr>
<tr>
<td>2.13</td>
<td>Auto-tuning time using EvolveTile with gcc</td>
</tr>
<tr>
<td>2.14</td>
<td>Auto-tuning time using EvolveTile with icc</td>
</tr>
<tr>
<td>2.15</td>
<td>Speed-up of Tiled Code with gcc</td>
</tr>
</tbody>
</table>
4.5 A loop nest ................................................................. 84
4.6 Optimized loop nest for SCC ........................................... 92
4.7 Coherence API for conservative handling .......................... 92
4.8 An iterative loop with irregular data references .................... 93
4.9 An iterative loop with irregular data references for SCC system .. 95
4.10 An iterative loop ....................................................... 97
4.11 L1 data cache read misses (lower, the better). The L1 read miss ratios for HCC are also shown. .................... 102
4.12 Execution time (lower, the better) .................................. 103
4.13 Traffic on the system bus (lower, the better). Average number of words per cycle for HCC is also shown. .................... 105
4.14 L1 and L2 Cache Energy (lower, the better). The first bar shows HCC energy and second bar SCC-opt energy .................. 106
5.1 Illustration of insertion of error detection codes .................... 116
5.2 Resilient Code .......................................................... 117
5.3 Example affine code snippet ......................................... 118
5.4 The example code’s abstract syntax tree ............................ 121
5.5 Example affine code snippet with checksum augmentation ........ 124
5.6 Checksum-augmented example affine code with index-set splitting optimization .................................................. 125
5.7 Code snippet illustrating the general checksum-augmentation scheme ... 127
5.8 Example code requiring dynamic use count determination ........ 129
5.9 Example code with inspector to determine use counts and the generalized checksum-augmentation scheme ........................................... 131

5.10 Normalized running time of the resilient codes with one checksum software-only solution. Absolute running times, in seconds, of the Original unmodified codes are also shown ...................................... 141

5.11 Estimated performance of resilient codes with a special function unit to compute checksums. y-axis employs a linear scale with selective tick marks to highlight the costs. .................................................... 144
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>24</td>
</tr>
<tr>
<td>2.2</td>
<td>28</td>
</tr>
<tr>
<td>2.3</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>59</td>
</tr>
<tr>
<td>3.2</td>
<td>61</td>
</tr>
<tr>
<td>3.3</td>
<td>69</td>
</tr>
<tr>
<td>4.1</td>
<td>100</td>
</tr>
<tr>
<td>4.2</td>
<td>101</td>
</tr>
<tr>
<td>5.1</td>
<td>138</td>
</tr>
<tr>
<td>5.2</td>
<td>140</td>
</tr>
<tr>
<td>5.3</td>
<td>153</td>
</tr>
<tr>
<td>5.4</td>
<td>156</td>
</tr>
<tr>
<td>5.5</td>
<td>157</td>
</tr>
<tr>
<td>5.6</td>
<td>158</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

The number of transistors on integrated chips has been doubling approximately every two years according to the so-called Moore’s law. Computer architects in the past would increase microprocessor clock frequency to translate higher transistor count to enhanced processor performance. But, simply increasing clock rate to achieve higher performance with each processor generation became impossible in the last decade because of unmanageable power dissipation on the chip. The hardware designers then introduced parallelism in hardware as a way to deliver higher performance rather than boosting performance of a single processor. Developing efficient software for parallel processing units — multi-core, and many-core processors has therefore become an important and urgent challenge to address for the academia, and industry alike.

Compiler Algorithms for High-Performance Parallel Code Generation. An attractive solution to the problem of developing parallel software is for a compiler to automatically generate high performing parallel code from a sequential code so that the programmer is freed from that burden. My research has addressed this problem in the context of loop-based scientific programs (Chapter 2): Tiling (also called blocking) of loops is an effective program transformation to obtain parallelism without incurring a lot of communication overhead (tiling reduces the volume, and frequency of communication between processors),
and with better cache locality. But, the selection of optimal tile sizes for tiled programs is critical to realize benefits of tiling. The compiler algorithms that we have developed generate *adaptive tiled codes*, where tile sizes can be changed on-the-fly during execution. This allows a rapid exploration of tile sizes to select the best tile sizes, and also helps tailor tile sizes to adapt software to changing hardware parameters, such as available cache capacity, and communication, and memory bandwidth.

**Achieving Energy Efficiency.** The energy, and power consumption along with performance have in the recent years become chief concerns for hardware, and software design for several reasons. First, in the future, even though a larger number of transistors can be fitted on a chip due to continued technology scaling, not all of them can be powered on simultaneously because of strict power limits, giving rise to “dark silicon”. Second, data centers consume a significant amount of energy, and that results in large carbon footprint, and energy bills. Therefore, developing measures in hardware, and software to reduce energy consumption have assumed significance.

We have attacked the energy consumption problem in two directions: 1) We have developed compiler analysis techniques to reduce energy expenditure in cache memories (Chapter 3); 2) We have architected an integrated hardware-software solution for compiler-directed cache coherence (Chapter 4) (instead of delegating the task of maintaining cache coherence in multi-processors to hardware as is the current state-of-practice).

**Cache Memory Energy Reduction.** On current processors, nearly half of the chip area is dedicated to last-level caches (LLCs). However, the entire available cache capacity may not be utilized by all applications as it depends on data reuse characteristics of the program at hand. There are only discrete cache sizes at which applications show marked change in
their miss rates. We developed compiler algorithms to analyze data reuse in applications to precisely identify such transition points where an application shows a change in miss rate. Then, the techniques developed output the useful cache size for a given system configuration. The fraction of the cache that is deemed not useful, can be power-gated to save leakage power of cache. The experimental evaluation revealed that, for most applications significant cache-power reductions can be obtained using the compiler analysis developed.

Software Cache Coherence. On multi-processor systems that have private caches, maintaining cache coherence in hardware (via snoopy bus protocols, or directory based protocols) faces scaling problems. A scalable alternative is to transfer the task of achieving cache coherence to software: cache coherence instructions, such as invalidate, and write-back are introduced in programs so that when a processor reads a variable, it is guaranteed to be seeing the value written by the most recent assignment to that variable. We developed compiler algorithms leveraging sophisticated dependence analysis capabilities of the Polyhedral model to automatically instrument programs with coherence instructions to correctly run them on software managed caches. We also proposed code transformations to reduce coherence traffic. Experimental results indicated that the hardware-software co-design that we proposed results in energy savings, and also is able to match/improve over performance of hardware coherence schemes.

Fault Tolerance against Memory Errors. The decreasing transistor sizes, use of lower voltage levels, and smaller noise margins have increased the probability of multi-bit errors in the memory subsystem. Hence, it is of increased interest to design efficient solutions to address this problem in software as hardware does not typically have embedded multi-bit error detection and correction mechanisms. Towards that end, we developed novel compiler
techniques to instrument application programs with error detection codes that at runtime protect every memory reference via the use of *checksums* (Chapter 5). The experimental evaluation demonstrates that the proposed solutions have low overheads and are practical.
Chapter 2: Adaptive Parallel Tiled Code Generation and Accelerated Auto-tuning

2.1 Introduction

Tiling [60, 123, 103, 98, 16, 127, 14] is an important loop transformation used to enhance temporal reuse of data and to expose and exploit parallelism. Tiling reschedules computations of a loop to make efficient use of a faster memory (such as L1 or L2 cache) in order to avoid high data access costs to a slower memory (such as RAM). Tiling for parallelism maps tiles to different processors where tiles are executed atomically and communication between processors occurs to exchange data only at the boundaries of tiles, if required. This reduces the frequency and volume of inter-processor communication leading to scalable parallelism.

The selection of correct tile sizes is however essential to ensure that tiling delivers the desired high performance since there can be a significant difference in the performance of tiled code with good and bad tile sizes. A vast body of research has tried to address the problem of tile size selection by analytically modeling execution time as a function of tile sizes, machine characteristics (e.g., cache size, cache associativity, TLB size, network latency, etc.) and program characteristics (e.g., data footprint, data reuse pattern, etc.) [124, 43, 13, 70, 28, 40, 4, 101, 105]. However, due to increasing architectural complexities
such as deep memory hierarchies, prefetching and variable inter-processor communication costs in multi-core machines, none of the analytical solutions have proved to be accurate across a wide array of target architectures and programs.

In response, empirical search based *auto-tuning* approaches [11, 112, 115, 68, 121] have been proposed where tiled code is executed on the target machine with a large number of tile sizes according to a chosen heuristic and then the best performing tile sizes found are designated to be used for the subsequent runs on the machine. A major concern for auto-tuning systems is the cost of pre-production runs which is expected to pay off once a sufficiently large number of “production” runs follow. The auto-tuning cost is easily justified for heavily used library routines but it may prove to be prohibitive for an extensive array of user-generated codes in a number of scientific application domains such as computational fluid dynamics, mechanical simulations, quantum chemistry, linear algebra and others.

In this chapter, we develop adaptive parallel tiled code generation techniques so that tile sizes may be modified on-the-fly during execution of the tiled code. Such a capability allows exploration of several tile sizes in a single run of the application and thus, accelerates the auto-tuning process. Adaptive tiling is orthogonal to the choice of auto-tuning algorithm and can be used in conjunction with any of the empirical search based schemes. Experimental evaluation using a state-of-the-art auto-tuning algorithm demonstrates that adaptive tiling can achieve up to 5X reduction in auto-tuning time.

In addition to its utility in accelerating auto-tuning, adaptive tiling is also useful for adaptive execution environments where architectural parameters may change on the fly during execution of a long running program. In future processors it is likely that, available cache for an executing program might change dynamically at runtime. This in turn means
that the best tile sizes can change, since the optimal value of tile sizes is directly affected by the available cache capacity. The code generation strategies developed in the chapter enable adapting tile sizes to changing execution environment and thereby, facilitate efficient execution in such contexts.

The number of transistors on a chip has been growing in accordance with Moore’s law (doubling every eighteen months) [81]. However, due to increase in leakage energy with decreasing feature sizes of transistors [66] and the end of Dennard scaling [33], it is no longer possible to maintain constant power density with successive generations of VLSI technology. Therefore for future technologies, it is expected that all transistors on the chip cannot be simultaneously powered on. According to a study conducted by Esmaeilzadeh et. al, at 8nm processor technology, more than 50% of a fixed-size chip must be powered-off because of strict power limits [39].

At the same time, on-chip caches account for a significant percentage of processor die area (40% or more on current processors) [5] and thus contribute to a significant fraction of the total processor power-consumption. In future processors, portions of configurable caches may be selectively turned off to save power. The power saved by reducing active cache size may be channeled instead to power on a “dark” core. Thus, depending on available parallelism in the prevailing workloads, processing cores may be turned on or off dynamically with a corresponding decrease or increase of the active sizes of caches on chip. With the use of adaptive parallel tiled code, tile sizes may be suitably modified as the cache size changes so that the best possible performance for the new execution environment is realized.

The chapter makes the following contributions:
• Techniques to generate correct and efficient adaptive parallel tiled code are developed.

• The benefit of adaptive tiling in accelerating auto-tuning process is experimentally demonstrated with the use of a state-of-the-art auto-tuning algorithm.

The rest of the chapter is organized as follows. Section 2.2 provides background material on parametric tiling. In Sections 2.4 and 2.5, adaptive parallel tiled code generation schemes are developed for two possible kinds of parallel tiled codes that may result depending on the data dependencies in the program being tiled. Section 5.7 details the experimental evaluation conducted. Related work is discussed in Section 4.7 and the chapter is concluded in Section 5.9.

2.2 Background

2.2.1 Parametric Tiling

Tile sizes in parametrically-tiled codes are symbolic constants that can be set at runtime. Parametric tiling is especially useful for auto-tuning systems as the same tiled code can be run with different tile sizes without having to re-generate the code (as opposed to tiled codes where tile sizes are fixed at the time of code generation).

Consider a loop nest of depth $n$ with loop iterators $v_1, \ldots, v_n$ from the outermost to the innermost loop in that order. Let the parametric one-level tiled code for the considered loop nest be generated by a parametric-tile-code-generator such as the PTile system [8]. It will have $n$ tile loops with iterators $t_1, \ldots, t_n$ and $n$ intra-tile loops with iterators $u_1, \ldots, u_n$. Let the tile sizes associated with tile loops with iterators $t_1, \ldots, t_n$ be $s_1, \ldots, s_n$, respectively.

Each iteration point in the iteration domain of the original loop nest with loop iterators $v_1, \ldots, v_n$ is identified by its iteration vector $(v_1, \ldots, v_n)$. An iterator $v_i$ is related to the
corresponding tile iterator $t_i$ and tile size $s_i$ by the following relation:

$$v_i = s_i t_i + u_i \land 0 \leq u_i \leq s_i - 1. \quad (2.1)$$

Therefore, for an iterator $v_i$ in the original program, its tile coordinate $t_i$ is the quotient when $v_i$ is divided by the tile size $s_i$.

$$t_i = \left\lfloor \frac{v_i}{s_i} \right\rfloor. \quad (2.2)$$

**Example:** Consider the loop-nest shown in Fig. 2.1. The parametric tiled code for the same is presented in Fig. 2.2. The tile loops are loops with iterators $t1$ and $t2$; the intra-tile loops are those with iterators $u1$ and $u2$. The values of loop iterators $v1$ and $v2$ are then computed from tile and intra-tile iterator values. If the values of the loop iterators thus calculated are within the original bounds of the loops then the program statements are executed.

The tile loop bounds are computed by projecting out the intra-tile iterators and this leads to the possibility of the iterator values derived from tile and intra-tile iterators being outside of the original iteration space for certain tile sizes [8]. Hence, a check to ensure that the iteration point is indeed within the original iteration domain is performed. Let us call this check the *original-bounds-check*.

The tiled iteration space is shown in Fig. 2.5. The two loops of the loop nest in Fig. 2.1 are depicted using two axes in the figure. Execution of the tiled code corresponds to partitioning of the iteration space into tiles (of tile sizes $s1$ and $s2$) and traversing within each tile (with intra-tile loops) and then traversing between tiles (using tile loops).

Instead of performing the *original-bounds-check* for each iteration point, one could merge the check with the computation of loop iterator values and replace intra-tile iterators with loop iterators themselves. Fig. 2.3 shows the tiled code after applying the said
optimization. The intra-tile loop iterators \( u_1 \) and \( u_2 \) do not explicitly appear in the code, rather they implicitly appear in the bounds of loop iterators \( v_1 \) and \( v_2 \). We refer to these optimized inner loops from now on as intra-tile loops.

### 2.2.2 Parallel Parametric Tiling

#### Parallel Tile Loops

In the parametric tiled code, if any of the tile loops is parallel (i.e., has no loop-carried dependencies) then parallel execution of tiles is directly possible.

**Example:** If the outer tile loop of the tiled code shown in Fig. 2.3 is parallel then parallel tiled code generated using OpenMP work-sharing constructs is shown in Fig. 2.7.

#### No Parallel Tile Loops - Wavefront Parallel

Even when none of the tile loops is parallel, wavefront (pipeline) parallel execution of tiles is still possible where the tiles are scheduled in *wavefronts* so that tiles in a *wavefront* may be executed in parallel [123].
The parallel tiled code with the wavefront loop generated by the PTile algorithm [8] will have a sequential wavefront loop to iterate over different wavefronts, enclosing \( n \) parallel tile loops with iterators \( t_1, \ldots, t_n \).

The wavefront loop variable is the sum of all the tile iterators:

\[
 w = \sum_{i=1}^{n} t_i. 
\]  

**Example:** The wavefront parallel tiled code corresponding to the loop nest in Fig. 2.1 is presented in Fig. 2.9. The wavefront loop variable is sum of tile iterators: \( w = t_1 + t_2 \). The lower bounds of tile iterators \( t_1 \) and \( t_2 \) (lb\_t1 and lb\_t2) are \( \text{ceil}(-1+1/s_1) \) and \( \text{ceil}(-1+1/s_2) \) respectively (in the sequential tiled code shown in Fig. 2.2). The upper bounds of \( t_1 \) and \( t_2 \) (ub\_t1 and ub\_t2) are \( \text{floor}(N/s_1) \) and \( \text{floor}(N/s_2) \) respectively. The possible values of the sum \( t_1 + t_2 \) range from \( \text{lb}\_t1 + \text{lb}\_t2 \) to \( \text{ub}\_t1 + \text{ub}\_t2 \). Therefore, the wavefront loop - \( w \) which is sum of \( t_1 \) and \( t_2 \), iterates from \( \text{lb}\_t1 + \text{lb}\_t2 \) to \( \text{ub}\_t1 + \text{ub}\_t2 \) (\( \text{floor}(N/s_1+N/s_2) \)). The inner two tile loops (\( t_1 \) and \( t_2 \) loops) find all tiles belonging to a particular wavefront.

The \( t_1 \) loop is related to the wavefront loop variable and \( t_2 \) by the relation: \( t_1 = w - t_2 \). Since, \( t_2 \) loop is inner to \( t_1 \) loop, value of \( t_2 \) cannot be used in the computation of \( t_1 \) bounds. Hence, value of \( t_2 \) is projected to either its lower-bound or to its upper-bound: The lower-bound of \( t_1 \) loop is maximum of its original lower-bound (\( \text{ceil}(1/s_1-1) \)) and \( w - \text{ub}\_t2 \) (\( \text{ceil}(w-N/s_2) \)). The upper-bound of \( t_1 \) loop is minimum of its original upper-bound (\( \text{floor}(N/s_1) \)) and \( w - \text{lb}\_t2 \) (\( \text{floor}(w-1/s_2+1) \)).

The \( t_2 \) loop is related to the wavefront loop variable and \( t_1 \) by the relation: \( t_2 = w - t_1 \). Since \( t_1 \) is outer to \( t_2 \) loop, the value of \( t_1 \) loop can be used in the computation of \( t_2 \) loop bounds. Thus, the lower-bound of \( t_2 \) loop is maximum of its original lower-bound
(ceil(1/s2-1)) and w-t1. The upper-bound of t2 loop is minimum of its original upper-bound (floor(N/s2)) and w-t1.

The execution of tiles in wavefronts is graphically shown in Fig. 2.11. The wavefront axes drawn - w = 0, w = 1 etc - pass through tiles belonging to those wavefronts. We note that even when the dependencies are along both the dimensions - v1 and v2, all the tiles in a wavefront are still parallel (i.e., they do not have dependencies between them) and after a wavefront is executed, all the tiles in the next wavefront are ready for execution.

2.3 Tile Size Adaptation for Sequential Tiling

Consider the problem of generating adaptive sequential tiled code, i.e., tiled code such that after a certain iteration of the outer tile loop, tile sizes can be changed and the resulting tiled execution is correct. The main code generation issue is that of ensuring that exactly the set of remaining iterations are executed after the tile sizes have been dynamically changed: every remaining iteration point must be scanned and no iteration point is scanned more than once.

For example, let us suppose that in the tiled code shown in Fig. 2.3, at the end of two iterations of the outer tile loop with iterator t1, tile sizes s1 and s2 are to be changed to s1.new and s2.new. At this point, the remainder of the iteration space can be characterized by inequality \(v1 \geq 2 \times s1\). The description of the iteration space along dimension v2 remains unchanged, which is, \(0 \leq v2 \leq N\). The scenario is graphically illustrated in Fig. 2.5.

Thus, to adapt tile sizes and to correctly scan the remaining iterations (unscanned iteration space), a mechanism is needed to dynamically set the lower bound for the outer loop at the time of tile-size-change. This can be achieved by introducing a new inequality \(v1 \geq v1.offset\) for the outer loop variable into the original description of the iteration.
\[ lb_{t1} = \text{ceil}(-1 + 1/s1) \]
\[ ub_{t1} = \text{floor}(N/s1) \]

// Tile loops
for (t1 = lb_{t1};
     t1 <= ub_{t1}; t1++) {
    for (t2 = ceil(-1 + 1/s2);
         t2 <= floor(N/s2); t2++)
       // Intra-tile loops
       for (v1 = max(s1 * t1, 0);
            v1 <= min(s1 * t1 + s1 - 1, N); v1++)
         for (v2 = max(s2 * t2, 0);
              v2 <= min(s2 * t2 + s2 - 1, N); v2++)
             S; // Statement
}

Figure 2.3: (Optimized) Sequential Tiled Code

\[ v1_{\text{offset}} = 0; \]
\[ lb_{t1} = \text{ceil}((v1_{\text{offset}} - s1 + 1)/s1); \]
\[ ub_{t1} = \text{floor}(N/s1); \]

// Tile loops
for (t1 = lb_{t1};
     t1 <= ub_{t1}; t1++) {
    for (t2 = ceil(-1 + 1/s2);
         t2 <= floor(N/s2); t2++)
       // Intra-tile loops
       for (v1 = max(s1 * t1, v1_{\text{offset}});
            v1 <= min(s1 * t1 + s1 - 1, N); v1++)
         for (v2 = max(s2 * t2, 0);
              v2 <= min(s2 * t2 + s2 - 1, N); v2++)
             S; // Statement
       if (\text{time\_to\_adapt}() == \text{true})
       {
        v1_{\text{offset}} = s1 * t1 + 1;
        s1 = s1_{\text{new}}; s2 = s2_{\text{new}}; // New tile sizes
        t1 = \text{ceil}((v1_{\text{offset}} - s1 + 1)/s1);
        // New lower bound
        ub_{t1} = \text{floor}(N/s1); // New upper bound
        t1 = t1 - 1; // due to 'for' loop increment
       }
}

Figure 2.4: Adaptive Sequential Tiled Code

Figure 2.5: Original Tiled Space

Figure 2.6: New Tiled Space
space. $v1_{\text{offset}}$ is a parameter and is initialized to the original lower bound of the loop variable $v1$. At the end of a particular iteration of the tile loop $t1$ when the tile sizes are to be adapted, iteration points up to $(s1 \ t1 + s1 - 1)$ of the outer dimension would have been scanned because of Equality 2.1, where $s1$ is its tile size. Therefore, $v1_{\text{offset}}$ is set equal to the first iteration point of the unscanned iteration space, which is $s1 \ t1 + s1$, at the time of tile-size-change and the lower bound and upper bound of the tile iterator $t1$ are re-evaluated. This guarantees that, only the remaining iterations are scanned after the tile sizes are dynamically changed.

The adaptive tiled code corresponding to the tiled code shown in 2.3 is outlined in Fig. 2.4. Fig. 2.6 sketches the tiled space with the old and new tile sizes. According to C language semantics, the statement $t1++$ gets executed at the end of an iteration of the $\text{for}$ loop. To counteract the increment statement during tile size change, the statement $t1=t1-1$ is inserted to be executed at the time of tile-size-change. This ensures that execution of the $t1$ loop starts from the re-computed lower bound of $t1$ and not from a value one more than that.

### 2.4 Tile Size Adaptation for Outer-loop Parallel Tiling

Consider dynamic change of tile sizes in a parallel tiled code whose outer tile loop is parallel: after a stipulated iteration of the outer tile loop, tile sizes are to be modified. Since iterations of the outer loop are executing on different processors in parallel and there is no synchronization between them, it is not possible to ascertain if all the iterations prior to the stipulated iteration are completed. In the absence of such a confirmation, the remaining iteration space cannot be deterministically characterized, which is essential to adaptive tiled code generation.
This limitation is overcome by strip-mining the parallel outer loop (i.e., it is split into two equivalent loops) and making the outer strip-mined loop sequential and the inner strip-mined loop parallel. Such a strip-mining procedure introduces synchronization points during execution of the parallel loop and hence, the unscanned iteration space can be precisely characterized. Tile sizes may be adapted at synchronization points (i.e., at the completion of outer sequential iterations) using the adaptive sequential tiling techniques described in §2.3. In order not to lose parallelism due to the strip-mining of the loop, loop-length of the inner parallel loop is set to a value equal to or a multiple of the number of available parallel threads.
The technique expounded above when applied on the code in Fig. 2.7 would result in the adaptive tiled code shown in Fig. 2.8.

An alternate approach enabling such a tile-size adaptation for outer-loop parallel tiled codes is to allocate a set of consecutive parallel loop iterations to each of the available threads (block-distributed) and then letting each thread adapt tile sizes private to itself. This method is more efficient compared to the approach where the parallel loop is strip-mined, in that this approach does not require synchronization between threads. However, if the objective for the use of adaptive tiled code is to discover effective tile sizes for performance then, we found in our experiments that the thread-private method discovers tile sizes which are inferior to those found via coordinated synchronized adaptation across threads as described earlier.

The reason for this phenomenon is that when threads adapt tile sizes in a non-coordinated fashion, because of shared resources in the system (shared last level cache, memory bandwidth etc.), their optimizations interfere with each other (different threads contend for the same resources and attempt to optimize local performance even at the expense of lowering performance of other threads) and they choose different local-optimal tile sizes and none of them may be optimal when used for the whole tiled code.

In coordinated parallel adaptation, interference is avoided by testing the same tile sizes for all the threads simultaneously and this method correctly chooses optimal tile sizes for the tiled code.

### 2.5 Tile size Adaptation for Wavefront Parallel Tiling

In the tiled code, if none of the tile loops are parallel, parallel execution of tiles can still be achieved by pipeline (or wavefront) scheduling of tiles as described in §2.2.2. An
important class of programs - *stencils* fall into this category of parallelism. Stencils are the compute-intensive loops in computer codes of computational fluid dynamics, partial differential equation solvers, image processing and other scientific and engineering applications.

Stencils are iterative computations and stencil operations involve updating of array elements using values from their neighboring array elements. Thus, data dependencies exist in all dimensions of the iteration space and as a result, parallel tile loops cannot be found. Therefore, parallel execution is realized though wavefront scheduling of tiles.

### 2.5.1 Tile Size Adaptation Challenges

The wavefront parallel tiled code for the 2D loop nest of Fig. 2.1 is shown in Fig. 2.9. The wavefront scheduling of tiles is graphically depicted in Fig. 2.11. The tiles along a *wavefront* are executed in parallel. Suppose the tile sizes are to be changed after the execution of wavefront 3, which is designated as $w_{\text{changeover}}$.

In sequential tiled codes, it is possible to characterize the *unscanned iteration space* (i.e., the set of iteration points yet to be executed) as a lower-bound for a dimension of the iteration space at the time of tile-size-change and this property is used in the generation of adaptive sequential tiled code. For example, in sequential tiling shown in Fig. 2.5, the unscanned iteration space is described by $v1 \geq 2s1$.

In wavefront parallel tiled codes however, the unscanned iteration space cannot be captured as a lower bound for any of the dimensions of the iteration space. For the tiled space delineated into wavefronts shown in Fig. 2.11, a lower-bound characterization cannot be made: inequalities $v1 \geq 4s1, v1 \geq 3s1, v1 \geq 2s1, v1 \geq s1$ and $v1 \geq 0$ are all true at different values of $v2$. Further, the remaining iterations are not amenable to characterization using
lb_w = ceil (1/s1 + 1/s2 - 2);
ub_w = floor (N/s1 + N/s2);

// Sequential wavefront loop
for (w=lb_w; w<=ub_w; w++) {
    lb_t1 = max (ceil (w-N/s2), ceil (1/s1 - 1));
    ub_t1 = min (floor (w-1/s2+1), floor (N/s1));
    // Tile loops
    #pragma omp parallel for [clauses]
    for (t1=lb_t1; t1<=ub_t1; t1++)
        for (t2=max (w-t1, ceil (1/s2 - 1)); t2<=min (w-t1, floor (N/s2)); t2++)
            // Intra-tile loops
            for (v1=max (s1*t1, 0); v1<min (s1*t1+s1-1, N); v1++)
                for (v2=max (s2*t2, 0); v2<min (s2*t2+s2-1, N); v2++)
                    S; // Statement
}

Figure 2.9: Wavefront Parallel Tiled Code

lb_w = ceil (1/s1 + 1/s2 - 2);
ub_w = floor (N/s1 + N/s2);
w_full = lb_w;

// Sequential wavefront loop
for (w=lb_w; w<=ub_w; w++) {
    lb_t1 = max (ceil (w-N/s2), ceil (1/s1 - 1));
    ub_t1 = min (floor (w-1/s2+1), floor (N/s1));
    // Tile loops
    #pragma omp parallel for [clauses]
    for (t1=lb_t1; t1<=ub_t1; t1++)
        for (t2=max (w-t1, ceil (1/s2 - 1)); t2<=min (w-t1, floor (N/s2)); t2++)
            // Intra-tile loops
            for (v1=max (s1*t1, 0); v1<min (s1*t1+s1-1, N); v1++)
                for (v2=max (s2*t2, 0); v2<min (s2*t2+s2-1, N); v2++)
                    if (w == w_full || (v1/s1_old+v2/s2_old) >= (w_changeover+1))
                        S; // Statement
            if (w == w_full & & time_to_adapt() == true) {
                w_changeover = w;
                s1_old = s1; s2_old = s2; // Save old tile sizes
                s1 = s1_new; s2 = s2_new; // New tile sizes
                Update w_start, w_full;
                ub_w = floor (N/s1+N/s2); // New upper bound
                w = w_start - 1; // due to 'for' loop increment
            }
}

Figure 2.10: Adaptive Wavefront Parallel Tiled Code

any affine expression as the unscanned space is non-convex as may be ascertained from the diagram.

We describe our solution to this adaptive code generation problem next.

2.5.2 Overview of the Solution

We generate wavefronts for the whole iteration space using new tile sizes and then superimpose the new wavefronts with the extant wavefronts. However, the superimposition may not be one-to-one in the sense that a wavefront in the original tiled space may straddle many wavefronts in the new tiled space. For example, consider wavefronts in a tiled space depicted in Fig. 2.11. Now, let the tile size along dimension v1 be increased three-fold and
the resulting wavefronts are shown in Fig. 2.12. Tiles of wavefront 3 in Fig. 2.11 \((w = 3)\) belong to 3 wavefronts in the new space: \(w' = 1\), \(w' = 2\) and \(w' = 3\).

Therefore, the first unscanned wavefront \(w_{\text{changeover}} + 1\) gives rise to a sequence of wavefronts in the new tiled space because of non one-to-one mapping of wavefronts with old and new tile sizes. Let these wavefronts be collectively referred to as the changeover region. Thus, execution after tile-size-change may be resumed from the wavefront that starts the changeover region (referred to as \(w_{\text{start}}\) in §2.5.3). But, until we get past the changeover region, we may find some scanned (or executed) iteration points. To avoid scanning them again, we need to check for the scanned status of the iteration points in that region (scanned check in §2.5.3). From the first wavefront that lies outside of the changeover region (referred to as \(w_{\text{full}}\) in §2.5.3), we are guaranteed that we have only unscanned points.
For the new tiled space shown in Fig. 2.12, execution is begun from wavefront 1 (marked as w\_start) which contains unscanned iterations. Wavefronts between 1 and 3 \((w' = 1, w' = 2 \text{ and } w' = 3)\) contain some scanned iteration points from old tile sizes. Thus, *scanned check* is operative in those wavefronts to ensure that no iteration point is doubly scanned. From wavefront 4 onwards (marked w\_full), we have only unscanned points. Hence, the *scanned check* is dropped when execution reaches wavefront 4.

### 2.5.3 Code Generation

**Scanned Check**

Let us assume that the tile sizes are to be changed from \(s_1, \ldots, s_n\) to \(s'_1, \ldots, s'_n\) in that order at the end of a certain iteration of the wavefront - \(w\) - loop (i.e., when the value of \(w\) is \(w_{changeover}\)). Further, let us call this stage during the program execution, the *changeover point*.

The iteration points whose tile coordinates sum up to \(w_{changeover}\) or less are scanned until the *changeover point* because of Equality (2.3). Therefore, an iteration point is scanned if its tile coordinators are such that,

\[
\sum_{i=1}^{n} t_i \leq w_{changeover}. \tag{2.4}
\]

From Equations (2.2) and (2.4), an iteration point \((v_1, v_2, \ldots, v_n)\) has been scanned prior to the *changeover point* if it satisfies the following condition:

\[
\sum_{i=1}^{n} \left| \frac{v_i}{s_i} \right| \leq w_{changeover}. \tag{2.5}
\]

Equation 2.5 is referred to as the *scanned check*. 

20
First non-empty wavefront and first full wavefront

- We wish to find the first wavefront below which we are guaranteed to not have any unscanned iteration points with the old tile sizes. Let this wavefront be denoted by \( w_{\text{start}} \). The execution is resumed from wavefront \( w_{\text{start}} \) after the tile sizes are modified.

- We seek to find the first wavefront at which and beyond which we are guaranteed to have only the unscanned iteration points. Let this wavefront be called \( w_{\text{full}} \).

Between wavefronts \( w_{\text{start}} \) and \( w_{\text{full}} - 1 \), the scanned check is used to ensure that a scanned iteration point is not scanned again: statements associated with an iteration point are executed only when the scanned check at that iteration point evaluates to false.

In this section, we derive expressions for \( w_{\text{start}} \) and \( w_{\text{full}} \) in terms of the wavefront value at the changeover point - \( w_{\text{changeover}} \), the old tile sizes - \( s_1, \ldots, s_n \) and the new tile sizes - \( s'_1, \ldots, s'_n \).

Let the scaling factors for tile sizes be \( \alpha_1, \alpha_2, \ldots, \alpha_n \) and each \( \alpha_i > 0 \): \( s'_1 = \alpha_1 s_1, s'_2 = \alpha_2 s_2, \ldots, s'_n = \alpha_n s_n \). Further, let \( \alpha_{\text{min}} = \min_{1 \leq i \leq n} \alpha_i \) and \( \alpha_{\text{max}} = \max_{1 \leq i \leq n} \alpha_i \).

An iteration point is identified by a vector of loop iterators: \( (v_1, v_2, \ldots, v_n) \). Tile coordinates in the old tiled space (with tile sizes \( s_1, \ldots, s_n \)) are denoted by \( t_1, \ldots, t_n \) and tile coordinates in the new tiled space (with tile sizes \( s'_1, \ldots, s'_n \)) are denoted by \( t'_1, \ldots, t'_n \). The wavefront loop variable in the old tiled space is \( w \) and in the new tiled space, it is \( w' \).

**Lemma 2.5.1.** For a given loop iterator \( v_i \),

\[
\frac{t_i}{\alpha_i} - 1 \leq t'_i \leq \frac{t_i + 1}{\alpha_i}
\]

21
Proof. Because of relation (2.1),

\[ t_i s_i \leq v_i \leq t_i s_i + s_i - 1 \quad \text{and} \quad t'_i \alpha_i s_i \leq v_i \leq t'_i \alpha_i s_i + \alpha_i s_i - 1 \]

\[ \Rightarrow t_i s_i \leq t'_i \alpha_i s_i + \alpha_i s_i - 1 \Rightarrow t'_i \geq \frac{t_i}{\alpha_i} - 1 + \frac{1}{\alpha_i} s_i - 1 \]

Also, \( t'_i \alpha_i s_i \leq t_i s_i + s_i - 1 \Rightarrow t'_i \leq \frac{t_i + 1}{\alpha_i} - \frac{1}{\alpha_i} \leq \frac{t_i + 1}{\alpha_i} \\

\[ \Rightarrow t_i s_i \leq t'_i \alpha_i s_i + \alpha_i s_i - 1 \Rightarrow t'_i \geq t_i \alpha_i \]

Lemma 2.5.2.

\[ \frac{w}{\alpha_{\text{max}}} - n \leq w' \leq \frac{w + n}{\alpha_{\text{min}}} \]

Proof.

From Lemma 2.5.1, \( \frac{t_i}{\alpha_i} - 1 \leq t'_i \leq \frac{t_i + 1}{\alpha_i} \)

Therefore, \( \sum_{i=1}^{n} \left( \frac{t_i}{\alpha_i} - 1 \right) \leq \sum_{i=1}^{n} t'_i \leq \sum_{i=1}^{n} \left( \frac{t_i + 1}{\alpha_i} \right) \Rightarrow \sum_{i=1}^{n} t'_i \geq \left( \sum_{i=1}^{n} \frac{t_i}{\alpha_i} \right) - n \geq \frac{\sum_{i=1}^{n} t_i}{\alpha_{\text{max}}} - n \)

Similarly, \( \sum_{i=1}^{n} t'_i \leq \sum_{i=1}^{n} \left( \frac{t_i + 1}{\alpha_i} \right) \leq \frac{\left( \sum_{i=1}^{n} t_i \right) + n}{\alpha_{\text{min}}} \)

Hence, \( \sum_{i=1}^{n} t'_i \leq \frac{\left( \sum_{i=1}^{n} t_i \right) + n}{\alpha_{\text{min}}} \)

But, \( \sum_{i=1}^{n} t_i = w \) and \( \sum_{i=1}^{n} t'_i = w' \). Therefore, \( \frac{w}{\alpha_{\text{max}}} - n \leq w' \leq \frac{w + n}{\alpha_{\text{min}}} \)

Lemma 2.5.3.

\[ w_{\text{start}} = \left\lfloor \frac{w_{\text{changeover}} + 1}{\alpha_{\text{max}}} \right\rfloor - n \]

Proof. Of all the possible wavefronts that the first unscanned wavefront \( w = w_{\text{changeover}} + 1 \) in the old tiled space can get mapped to in the new tiled space, the smallest of those wavefronts - \( w_{\text{start}} \) - can be found as follows.

According to Lemma 2.5.2, for \( w = w_{\text{changeover}} + 1 \),

\[ w' \geq \frac{w_{\text{changeover}} + 1}{\alpha_{\text{max}}} - n \Rightarrow w' \geq \left\lfloor \frac{w_{\text{changeover}} + 1}{\alpha_{\text{max}}} - n \right\rfloor \]

22
Therefore, the first *wavefront* to have any unscanned iteration points in the new tiled space is \( \left\lfloor \frac{w_{\text{changeover}} + 1}{\alpha_{\text{max}}} - n \right\rfloor \).

**Lemma 2.5.4.**

\[
W_{\text{full}} = \left\lceil \frac{W_{\text{changeover}} + n}{\alpha_{\text{min}}} \right\rceil + 1.
\]

**Proof.** Of all the possible *wavefronts* that the last scanned *wavefront* \( W = w_{\text{changeover}} \) in the old tiled space can get mapped to in the new tiled space, the last of those *wavefronts* can be found as follows.

According to Lemma 2.5.2, for \( W = w_{\text{changeover}} \),

\[
w' \leq \frac{W_{\text{changeover}} + n}{\alpha_{\text{min}}} \Rightarrow w' \leq \left\lceil \frac{W_{\text{changeover}} + n}{\alpha_{\text{min}}} \right\rceil
\]

Therefore, *wavefronts* from wavefront \( \left\lceil \frac{W_{\text{changeover}} + n}{\alpha_{\text{min}}} \right\rceil + 1 \) onwards are guaranteed to contain only *unscanned* iteration points.

The adaptive wavefront parallel tiled code corresponding to the tiled code shown in Fig. 2.9 is presented in Fig. 2.10.

**Scanned Check at Tile Level**

Even when \( w_{\text{start}} \) and \( W_{\text{full}} \) are tight in that they only mark those *wavefronts* that contain a mix of scanned and unscanned points, there may still exist tiles in the *changeover region* which are either *fully scanned* or *fully unscanned*.

For example, for the tiled space shown in Fig. 2.12, in the *wavefront* marked \( w_{\text{start}} \), the bottom-right tile contains both scanned and unscanned iterations while the upper-left tile contains only the scanned iteration points. In the next wavefront - \( w' = 2 \), lower-right tile is completely unscanned while the upper-left tile is made up of both scanned and unscanned points. Therefore, we want to find a way to hoist the *scanned check* at the tile level so that
Table 2.1: Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>covcol</td>
<td>Covariance matrix computation</td>
</tr>
<tr>
<td>corcol</td>
<td>Correlation matrix computation</td>
</tr>
<tr>
<td>tmm</td>
<td>Triangular matrix multiplication</td>
</tr>
<tr>
<td>trisolv</td>
<td>Triangular solver</td>
</tr>
<tr>
<td>seidel</td>
<td>Linear equations solver - Gauss-Seidel method (a stencil)</td>
</tr>
<tr>
<td>adi</td>
<td>A partial differential equations solver (a stencil)</td>
</tr>
<tr>
<td>jacob-2d</td>
<td>Linear equations solver - Jacobi method (a stencil)</td>
</tr>
<tr>
<td>fdtd-2d</td>
<td>2-D Finite Difference Time Domain kernel (a stencil)</td>
</tr>
<tr>
<td>lu</td>
<td>LU decomposition</td>
</tr>
</tbody>
</table>

the *scanned check* is confined to exactly those tiles which have both scanned and unscanned iteration points.

By relation (2.1), \( t_i' s_i' \leq v_i \leq t_i' s_i' + s_i' - 1 \). Therefore if, \( \sum_{i=1}^{n} \left[ \frac{t_i' s_i' + s_i' - 1}{s_i} \right] \leq w_{changeover} \), the whole tile is scanned. And if, \( \sum_{i=1}^{n} \left[ \frac{t_i' s_i'}{s_i} \right] \geq w_{changeover} + 1 \), the whole tile is unscanned.

The adaptive parametric tiled code is generated with these optimizations.

### 2.6 Experimental Evaluation

We evaluate the effectiveness of adaptive tiling in accelerating auto-tuning for selection of high performing tile sizes. The auto-tuning time and the quality of tile sizes selected using non-adaptive and adaptive tiling schemes are compared.

#### 2.6.1 Benchmarks and Set-up

The benchmark programs used in the experiments and their descriptions are shown in Table 2.1.
For each benchmark, experiments were conducted with 3 problem sizes (array dimensions): 1000 (small), 3000 (medium) and 6000 (large). In stencil codes, the number of time loop iterations are set to 50, 500, and 1000 for small, medium, and large problem sizes respectively.

In the tiled codes of covcol, corcol, tmm, trisolv benchmarks, the outer tile loop is parallel. These benchmarks are used for evaluating effectiveness of adaptive outer-loop parallel tiling. In the tiled codes of other benchmarks - seidel, jacobi-2d, fdtd-2d, adi and lu, the outer tile loop is not parallel and parallelism in the tiled codes of those benchmarks is realized through wavefront scheduling of tiles. These benchmarks are used to evaluate efficacy of adaptive wavefront parallel tiling.

The experiments are run on dual Intel Xeon E5630 quad-core processors, running at 2.53GHz with 32KB L1 cache. The programs were compiled using gcc 4.6.2, with -O3 -fopenmp flags and usingicc 12.0.4, with -fast -openmp flags. The parallel tiled codes are run using 8 threads.

2.6.2 Methodology

Tile Size Selection Algorithm

The parallel rank ordering algorithm (PRO) [111] is a state-of-the-art auto-tuning algorithm and is used in this chapter for tile size selection. PRO is a simplex based method designed for faster convergence and is thus well suited for online auto-tuning where tile size selection is carried out along with performing of useful computation.

All the benchmarks considered have 3-D nested loops and hence all of them have 3 tile sizes to tune. The initial simplex in PRO is constructed around the point \((32,32,32)\) (the three tile sizes are \((32,32,32)\)). Thus, the centroid of the simplex \(-c\) is set to \((32,32,32)^T\). The number of vertices of the simplex is \(2N\) where \(N\) is the number of tile sizes to be tuned.
The vertices which form the initial simplex are: \[ \{(c + b_1.e_i), (c - 0.5b_i.e_i), i = 1, \ldots, N\} \]
where \( b_i = (32, 32, 32)^t \) as well.

**Auto-tuning Time Comparison**

The auto-tuning time of adaptive tiling (henceforth referred to as *EvolveTile*) is compared with that of a non-adaptive tiling scheme, PTile [8]. The same tile size selection algorithm PRO is used with both the tiling schemes, EvolveTile and PTile. We note that PTile is also a parametric tiling procedure. However, with PTile, tile sizes cannot be changed during execution; the tiled code has to execute with a set of tile sizes from beginning to the end of its execution.

The PRO algorithm informs PTile of the tile sizes whose performance is to be measured. When the PTile code completes a complete run with those tile sizes, it reports performance of the tile sizes back to PRO. PRO then provides the next tile sizes to explore or if the algorithm converges, it outputs the best performing tile sizes discovered. The total auto-tuning time before PRO algorithm converges is recorded.

**Tile Size Adaptation Frequency in EvolveTile**

In EvolveTile, tile sizes can be changed after any iteration of the outer tile loop. This fact may be used to observe performance of a few outer tile loop iterations and subsequently, tile sizes could be changed to the next tile sizes to be explored. Thus, in a single run, performance of a number of tile sizes may be tested and auto-tuning expedited. However, for the discovery of good tile sizes, it is also crucial that the few iterations when certain tile sizes are active, can indeed be used to reliably gauge performance of those tile sizes for the whole tiled code. Hence, the following technique is employed: 1) The number of iteration points for which certain tile sizes are to be kept active - *stable number of*
- is determined and then, 2) Per-iteration performance of those many iterations is observed and tile sizes are changed as indicated by PRO.

The stable number of iterations is arrived at as follows. When the per-iteration time of two consecutive sets of outer-tile-loop-iterations stabilizes, the number of iteration points contained in those outer tile iterations is designated as stable number of iterations. (The per-iteration time is derived by dividing the execution time of outer tile loop iterations with the number of iteration points contained in them.) Tile sizes are changed every stable number of iterations according to tile sizes given out by the PRO algorithm. The total auto-tuning time, until PRO converges is noted.

**Tile Size Performance**

The performance of tiled code with the following four sets of tile sizes is reported.

- **PTile PRO:** The performance of tiled code using the best tile sizes found via auto-tuning with PTile is reported.

- **EvolveTile PRO:** Performance of tile sizes discovered via auto-tuning using EvolveTile is shown.

- **PTile Exhaustive:** In addition, an exhaustive search for optimal tile sizes is carried out by running tiled code with power-of-two tile sizes between (2, 2, 2) and (1024, 1024, 1024) (1000 combinations of rectangular tile sizes). Performance of tiled code using tile sizes discovered via such an exhaustive search is also presented. This will help us assess the quality of tile sizes found via PRO.

- **Default:** The performance of default tile sizes (32, 32, 32) around whom the initial simplex is built in the PRO auto-tuning algorithm is reported as well.
Each benchmark program is run 5 times and the mean execution times of those runs are used for reporting.

2.6.3 Results

![Auto-tuning Time with gcc](image)

Figure 2.13: Auto-tuning time using EvolveTile with gcc

Table 2.2: Average Auto-tuning time and Average number of runs for auto-tuning

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Fraction of Auto-tuning Time (EvolveTile PRO time / PTile PRO time)</th>
<th>Number of runs of application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>gcc</td>
<td>icc</td>
</tr>
<tr>
<td>small (S)</td>
<td>0.27</td>
<td>0.31</td>
</tr>
<tr>
<td>medium (M)</td>
<td>0.23</td>
<td>0.30</td>
</tr>
<tr>
<td>large (L)</td>
<td>0.20</td>
<td>0.22</td>
</tr>
</tbody>
</table>
Figure 2.14: Auto-tuning time using EvolveTile with icc

Auto-tuning Time

The auto-tuning time using EvolveTile as a fraction of auto-tuning time using PTile (both of them are using PRO algorithm) for various benchmarks and 3 different problem sizes are shown in Fig. 2.13 and Fig. 2.14 for gcc and icc compilers respectively. Table 2.2 compiles average auto-tuning time fractions across benchmarks. EvolveTile auto-tuning time ranges from 0.31 to 0.20 as a fraction of PTile auto-tuning time. The table also presents average number of times a benchmark is run before auto-tuning completes for the three auto-tuning approaches: PTile Exhaustive, PTile PRO and EvolveTile PRO.

We observe that the ratio of the number of runs of EvolveTile code to that of PTile code for auto-tuning is smaller than the fraction of EvolveTile auto-tuning time. This corresponds to a certain overhead associated with tile size change during execution: the overhead resulting from the introduction of synchronization points in adaptive outer-loop
parallel tiled code and potential re-scanning of wavefronts in the changeover region in adaptive wavefront parallel tiled code.

The overhead of tile-size-change however decreases as the problem size increases. This can be observed from the fact that even though the number of runs of EvolveTile during auto-tuning remains relatively the same for different problem sizes, auto-tuning time fraction decreases as the problem size increases. This is because, in outer-loop parallel codes, the number of synchronization points (which is proportional to loop-length of the outer tile loop) increases only slowly compared to the total amount of work (which is proportional to the product of lengths of all loops - three loops for the considered benchmarks) as the problem size is increased. In wavefront parallel adaptive tiling, the number of wavefronts in the changeover region (wavefronts between $w_{start}$ and $w_{full} - 1$) remains the same as the problem size increases (as $w_{start}$ and $w_{full}$ expressions are independent of problem sizes) even as the total number of wavefronts increases. Hence, overhead of potential rescanning of changeover region decreases as a percentage of total work.

**Tile Size Performance**

Table 2.3: Average Speed-ups over un-tiled codes across Benchmarks in post auto-tuned runs

<table>
<thead>
<tr>
<th>Problem size</th>
<th>gcc Default</th>
<th>gcc PTile Exhst.</th>
<th>gcc PTile PRO</th>
<th>gcc EvolveTile PRO</th>
<th>icc Default</th>
<th>icc PTile Exhst.</th>
<th>icc PTile PRO</th>
<th>icc EvolveTile PRO</th>
</tr>
</thead>
<tbody>
<tr>
<td>medium (M)</td>
<td>10.72X</td>
<td>13.27X</td>
<td>12.74X</td>
<td>12.76X</td>
<td>15.47X</td>
<td>19.25X</td>
<td>18.21X</td>
<td>18.29X</td>
</tr>
<tr>
<td>large (L)</td>
<td>12.69X</td>
<td>15.35X</td>
<td>14.65X</td>
<td>14.54X</td>
<td>17.59X</td>
<td>21.19X</td>
<td>20.39X</td>
<td>20.43X</td>
</tr>
</tbody>
</table>
Figure 2.15: Speed-up of Tiled Code with gcc

Figure 2.16: Speed-up of Tiled Code with icc
The best tile sizes discovered via auto-tuning are then used to measure the speed-ups achieved by parallel tiled codes using 8 threads over sequential un-tiled codes: PTile code is run with the best tile sizes determined with exhaustive search and PRO; EvolveTile code is executed with the best tile sizes discovered with PRO. The performance of PTile code with default tile sizes - (32, 32, 32) is also presented. The speed-ups are shown in Fig. 2.15 and Fig. 2.16 for gcc and icc compilers respectively. The legends S, M and L in the figures represent Small, Medium and Large problem sizes in that order. Average speed-ups across benchmarks are presented in Table 2.3.

In these post auto-tuning runs, tiled codes are executed with the best tile sizes found via respective schemes (and tile sizes are not changed). In outer-loop parallel EvolveTile code, tile size of the outer strip-mined tile loop is set to its loop length (i.e., s1_outer = ub_t1_outer - lb_t1_outer+1 in Fig. 2.8) so that there is only one iteration of the outer strip-mined tile loop and hence, there are no synchronization points during execution. In wavefront parallel EvolveTile, there is no changeover region as tile sizes are kept constant and hence, no overhead due to re-scanning of wavefronts in the changeover region. Thus, EvolveTile has no overheads vis-à-vis PTile if there are no tile size changes.

The speed-ups achieved by tiled codes increase as the problem sizes are increased. Tiling enhances data reuse in cache and therefore decreases memory access time and demand for memory bandwidth. Larger problem sizes being more bottle-necked by these factors benefit from tiling even more than smaller problem sizes. Performance of the tile sizes found via auto-tuning using EvolveTile (EvolveTile PRO) closely matches with the performance of those found through auto-tuning using PTile (PTile PRO).

Exhaustive search for optimal tile sizes yields slightly better tile sizes in some cases but incurs a much higher cost compared to auto-tuning with PRO algorithm. For most
benchmarks however, (except for fdtd-2d and lu), the performance of PRO tile sizes equals performance of exhaustive search tile sizes.

The performance of default tile sizes, namely, (32, 32, 32) varies considerably across benchmarks. For corcol and covcol, default tile sizes achieve performance comparable to that of auto-tuned tile sizes. For seidel, adi, jacobi-2d, and fdtd-2d however, performance of default tile sizes is much worse than that of auto-tuned ones. The performance comparison of default tile sizes with that of auto-tuned tile sizes (PRO and exhaustive) underscores the imperative of auto-tuning to realize high performance.

2.7 Related Work

A number of cache-based performance models [124, 43, 13, 70, 28, 40, 105] have been used to decide Tile-Size-Selection (TSS) for sequential tiling. Analytical solutions to TSS for wavefront parallel tiling in the case of 2D iteration spaces [4] and stencil computations [101] have been proposed by modeling computation and communication time as functions of tile sizes, network parameters etc.

Many works [11, 112, 115, 68] including the ATLAS project [121] use a combination of heuristic and empirical search based approaches to selecting good tile sizes.

Some authors [113, 64] have developed adaptive sequential tiling for TSS, and also to overcome conflicts in shared cache [84] but these approaches are not applicable to parallel tiling. An on-line tile size and unroll factor selection technique is proposed [116] but it can only be used for programs with an outer time loop or parameter sweep loop that surrounds tiled code. In contrast, the approach developed in this chapter enables dynamic on-the-fly change of tile sizes.
2.8 Conclusion

Tiling is a key program transformation that enhances cache data locality and exposes coarse grain parallelism. In this chapter, we developed adaptive parallel tiling methods where tile sizes can be changed on-the-fly during execution. The solutions presented overcome a fundamental challenge to the generation of adaptive tiled code in parallel contexts.

Using adaptive tiling, several tile sizes can be tested for their performance in a single run and thus, auto-tuning for tile size selection can be accelerated. Experimental results demonstrate that significant reduction in auto-tuning time can be achieved with the use of the techniques developed in this chapter. We believe that the developed techniques will also be highly useful in environments where power can be saved by turning off portions of caches during execution.
Chapter 3: A Compiler Analysis to Determine Useful Cache Size for Energy Efficiency

3.1 Introduction

Beginning from the 1980s, processor and memory system performance have greatly diverged [57]. To bridge the gap between the rate at which data can be processed by the processor and the rate in which it can be supplied by the memory, multi-level memory hierarchy was introduced with faster memory units (caches) placed between the fast processor and the slow memory so that these caches may deliver reused data to the processor at a much higher rate than the memory system. The size of these caches has grown over the years and on contemporary machines, the last-level cache takes up a significant fraction of the processor die area (40% or more) [5].

With decreasing feature sizes of transistors, static power (also known as leakage power) consumption has risen to levels comparable to that of dynamic power and currently accounts for 20-40% of the total power consumption of the processor [66]. With the use on near-threshold voltage devices, static leakage power is expected to consume an even larger fraction of total power. The number of transistors on the chip is still growing in accordance with Moore’s law [81], but a growing fraction of the transistors cannot be simultaneously
powered on because of strict power limits [36]. Considering this, leakage power is particularly limiting and ways have to be found to mitigate the challenge.

An outcome of the combination of above two trends has been that large caches account for a sizable fraction of the total power-dissipation. This is especially of concern when the cache is not beneficially used: when the inherent characteristics of the application are such that increase of cache size does not result in a lower miss-rate. As shown later in the chapter, there are a few discrete values of cache size at which the miss rate of an application changes significantly, but stays relatively unchanged for cache sizes in between these discrete transition points. Therefore, in these scenarios, unusable cache capacity can be turned off without increasing execution time of the application.

A number of prior studies have made this observation and techniques have been proposed to save dynamic and leakage power consumption due to caches without adversely affecting or minimally affecting performance [96, 129, 37, 117, 110, 63, 51]. A common theme of these approaches is that they observe cache hit/miss rates of the applications and then resize cache based on the observations. The techniques work using hardware mechanisms or profile driven analysis to make resizing decisions. One issue with these dynamic schemes is that when a portion of cache is turned off, the data currently held in those portions of the cache must be saved in memory before the cells can be powered off.

In this chapter, we develop a compile-time analysis for affine computations (programs where the loop bounds and array references are affine functions of loop iterators and program parameters) to determine useful cache-size for a given system configuration. Affine computations form an important class of programs: the compute-intensive loops in many
scientific codes (computational fluid dynamics, adaptive mesh refinement, numerical analysis etc.) are affine. According to a study [9], over 99% of loops in 7 out of the 12 programs of the SpecFP2000 and Perfect Club benchmarks are affine loops.

Our analysis computes possible working sets corresponding to different data reuses and determines the size of cache that can be profitably used under the maximum available cache size constraint (defined by the system configuration). The algorithm also outputs the fractions of cache to be allocated to different arrays of the program. The selective partitioning of cache between different arrays can be accomplished using page coloring [114, 75], a software cache partitioning mechanism.

In contrast to previously proposed dynamic schemes, the compile-time determination implies that a fraction of the cache can be turned off before starting execution of the application, so that no saving of data to memory will be needed. Another advantage of compile time discovery of useful cache size is that the useful cache may be divided among different arrays according to their reuse characteristics using software page coloring techniques. Such a discriminating allocation of cache is more difficult or even impossible to achieve with dynamic schemes for cache resizing since the observed dynamic behavior would be based on standard cache mapping.

The chapter makes the following contributions:

• An accurate compile-time approach is developed to compute working-sets corresponding to data reuse in the program.

• An algorithm to compute the useful cache size for affine programs under a maximum cache size constraint is presented.
The effective-cache-capacity profile is generated for several benchmarks over a range of problem sizes, demonstrating that significant fractions of the L3 cache in current processors can be turned off without performance penalty.

The rest of the chapter is organized as follows. After motivating the work with an example in Section 3.2, Section 3.3 introduces the notation and the building blocks used in developing the compiler algorithm. Section 3.4 develops the compiler analysis to compute useful-cache-size. Section 5.7 details the experimental evaluation conducted. Related work is discussed in Section 4.7 and the chapter is concluded in Section 5.9.

3.2 Motivation

![Figure 3.1: The Core i7 die. Source: Intel, [30]](image)
Fig. 3.1 shows the picture of Intel Core i7 processor die. The major components are labeled. It has 4 cores and each core has 32KB L1 data cache, 256KB L2 cache. Additionally, there is an 8MB L3 cache shared among 4 cores. Close to half of the chip area is dedicated to the last-level (L3) cache.

To study the relation between the presence of L3 cache to the performance of applications we conduct the following experiment using Seidel stencil whose code is shown in Fig. 3.2. Seidel is used in the Gauss-Seidel method for iteratively solving a linear system of equations.

The code is executed on different L3 cache sizes using a cache simulator - Dinero [34]. The L3 cache size is varied from 1KB to 2MB (we assume that each of the four cores uses 2MB of total 8MB shared L3 cache). The experiment is performed for different problem sizes: TSTEPS is set to 200 and $N$ is varied from 100 to 1000. For each problem size, the amount of L3 cache that can be turned off without increasing cache miss rate is recorded. Fig. 3.3 plots the results of the exercise.

When $N$ is 100, 93.75% of 2MB cache may be turned off without degrading performance. At $N = 200$, percentage of reduced cache falls to 75% and as the problem size is further increased to $N = 400$, no amount of L3 cache may be turned off without adversely impacting performance. However, if the problem size is changed to 600, L3 cache may be

```c
for (t = 0; t <= TSTEPS-1; t++)
  for (i = 1; i <= N-2; i++)
    for (j = 1; j <= N-2; j++)
```

Figure 3.2: seidel stencil
shrunk to 99.22% of its original size with no performance loss. For problem sizes $N = 800$ through $N = 1000$, 98.44% of cache can be powered off.

The reason for the observed behavior of Seidel towards its cache-needs is that, reuse in cache for Seidel occurs if the cache is capacious enough to hold i) $3N$ or ii) $N^2$ elements of array $A$ of type double. That is, if $N^2$ elements can be placed in cache then reuse occurs in the outermost loop and also in the inner loops; otherwise, if only $3N$ elements can be accommodated in cache, reuse occurs in the inner to outermost loop - $i$ loop. From $N = 100$ to $N = 400$, 2MB cache is large enough to hold $N^2$ elements and remainder of cache may be turned off, which tapers off to zero at $N = 400$. But beyond that, $N^2$ elements cannot be fitted in cache and hence, reuse in the outermost loop cannot be realized. In such a scenario, keeping only that much of cache so as to house $3N$ elements will be enough to
realize reuse in \( i \) loop without increasing cache miss rate. The effect of which is seen from \( N = 600 \) onwards in the increased percentage of non-beneficial cache.

The experiment indicates that, for a given program, reuses occur at certain discrete values of cache size: if the available cache size is smaller than the cache required to realize a particular reuse then cache may be shrunk to just fulfill the needs of the next reuse and this, \textit{without} incurring any performance penalty. This chapter presents an algorithm to statically determine such values of cache sizes, referred to as \textit{useful cache sizes}.

### 3.3 Background

Before presenting the compiler algorithm for cache analysis, we describe the mathematical notation we use and background information on reuse vectors. The codes shown in Fig. 3.4, Fig. 3.5 and Fig. 3.6 are used as working examples to illustrate the notation and the compiler algorithm.

#### 3.3.1 Notation

We use the same notation as used by LaMielle et al. [71] for the definition of sets, relations, apply, inverse and compose operations.

```plaintext
for (t = 0; t < steps; t++)
{
    for (i1 = 0; i1 < n; i1++)
        for (i2 = 1; i2 < n; i2++)
            Access X[i1][i2], X[i1][i2-1];
}
```

Figure 3.4: A perfectly nested loop
Sets

\[ s = \{ [x_1, \ldots, x_m] : c_1 \land \cdots \land c_n \} \]

where each \( x_i \) is a tuple variable/iterator and each \( c_j \) is a constraint.

The iteration spaces of the statements can be represented as sets. For example, iteration space of the code in Fig. 3.4 can be specified as the set \( I \):

\[ I = \{ [t, i1, i2] : (0 \leq t < tsteps) \land (0 \leq i1 < n) \land (1 \leq i2 < n) \} \]

Relations

\[ r = \{ [x_1, \ldots, x_m] \mapsto [y_1, \ldots, y_n] : c_1 \land \cdots \land c_p \} \]

where each \( x_i \) is an input tuple variable, each \( y_j \) is an output tuple variable and each \( c_k \) is a constraint.

Array accesses appearing in the code may be modeled as relations from iteration spaces to the access functions of the array references. For example, the two accesses to array ‘X’ in Fig. 3.4 - \( X[i1][i2] \) and \( X[i1][i2-1] \) - are represented as the following relations:

\[ r_1 = \{ [t, i1, i2] \mapsto [i1', i2'] : (i1' = i1) \land (i2' = i2) \} \]
\[ r_2 = \{ [t, i1, i2] \mapsto [i1', i2'] : (i1' = i1) \land (i2' = i2 - 1) \} \]

The Apply Operation

The apply operation on a relation \( r \) and a set \( s \) produces a set \( s' \) denoted by, \( s' = r(s) \) and is mathematically defined as:

\[ (\vec{x} \in s') \iff (\exists \vec{y} \text{ s.t. } \vec{y} \in s \land (\vec{y} \mapsto \vec{x}) \in r) \]
The set of array elements accessed by an array reference in a loop (data-footprint) may be derived by applying access function relations on the iteration space sets. For the array accesses in the example shown in Fig. 3.4, data-footprints of the two accesses are: \( r_1(I), r_2(I) \).

**The Inverse Operation**

The inverse operation \( r = r_k^{-1} \) operates on a relation \( r_k \) to produce a new relation \( r \) such that \( r \) has the same constraints as \( r_k \) but with the input and output tuple variables swapped.

\[
(\vec{x} \mapsto \vec{y} \in r) \iff (\vec{y} \mapsto \vec{x} \in r_k).
\]

Given a particular array element, inverse operation may be used to find all the iterations in which it is used by forming an inverse of the access function relation and then applying the newly formed relation on the iteration set. For example, the iterations - set \( I_{2,2} \) - that access \( X[2][2] \) may be found using the following operations.

\[
PE = \{[2,2]\} \quad I_{2,2} = \{r_1^{-1}(PE) \cup r_2^{-1}(PE)\}
\]

\( I_{2,2} \) will be evaluated to \( \{[t,2,3], [t,2,2]\} \) where \( t \) is an existential.

**The Compose Operation**

The compose operation on two relations \( r_i \) and \( r_j \) - \( r = r_i(r_j) \) - is defined as,

\[
(\vec{x} \mapsto \vec{y} \in r) \iff (\exists \vec{z} : \vec{x} \mapsto \vec{z} \in r_j \land \vec{z} \mapsto \vec{y} \in r_i)
\]

The compose operation is legal only when the output arity of \( r_j \) is equal to the input arity of \( r_i \).
### 3.3.2 Polyhedral Reuse Distance Vectors

In the Polyhedral model [125, 41], iteration space of a loop nest of depth $n$ is modeled as a convex integer polyhedron $\mathbb{Z}^n$ whose bounds are derived from bounds of the loop. Each iteration in the loop corresponds to a point in the polyhedron and is identified by the iteration vector $\vec{p} = (p_1, p_2, \ldots, p_n)$, where $p_i$’s are the values of the iterators of the loop.

The vector difference between two iteration vectors $\vec{p}_1$, $\vec{p}_2$ denotes distance in the polyhedral space between the iterations corresponding to those iteration vectors and is referred to as the distance vector $\vec{d}$.

When in two different iterations of the loop, the same data element is used, the reuse can be specified as a distance vector between the two iterations. E.g., The code shown in Fig. 3.4 has 3 loops and therefore its iteration space is modeled by vectors of dimension 3. A particular $X[i_1][i_2]$ which is accessed in iteration $(t, i_1, i_2)$ is accessed again in $(t+1, i_1, i_2)$. The reuse distance vector for this reuse is $(1, 0, 0)$.

### 3.3.3 Different Reuses

There may exist the following 4 kinds of reuse in a loop.

- **Self-Temporal** reuse occurs when the same data location is referenced in different iterations. E.g., In Fig. 3.4, the reference $X[i_1][i_2]$ has a self-temporal reuse vector $\vec{d} = (1, 0, 0)$.

- **Self-Spatial** reuse occurs when the data elements accessed by a reference in different iterations are present in the same cache line. The reference $X[i_1][i_2]$ in Fig. 3.4 possesses self-spatial reuse and because of self-spatial reuse alone, the memory loads for this reference may be reduced by a factor of $L$, where $L$ is the number of elements of array $X$ that can fit in a cache-line.
• *Group-Temporal* reuse arises when different references refer to the same memory location in different iterations. In Fig. 3.4, group-temporal exists between references $X[i1][i2]$ and $X[i1][i2-1]$ with reuse vector $(0, 0, 1)$.

• *Group-Spatial* reuse takes place when different references refer to the same cache line.

The mathematical properties that can be used to automatically identify these reuses and calculate the corresponding reuse vectors may be found in Wolf and Lam’s treatise on data locality [125].

### 3.3.4 Working Set

The *reuse distance vector* expresses distance in the iteration space between two iterations $\vec{i}_1$ and $\vec{i}_2$ where the same data element is accessed. For the reuse to be realizable in a particular cache, the data element resident in cache during iteration $\vec{i}_1$ should not be evicted (from cache) at least before the iteration $\vec{i}_2$ is executed. Equivalently, all the array elements accessed between iterations $\vec{i}_1$ and $\vec{i}_2$ must be retained in the cache. We refer to this set of array elements as the *working set*. We elaborate on the calculation of *working set size* next.

### 3.4 Compiler Analysis of Working Sets

#### 3.4.1 Working Set Computation

In order to compute the *working set*, we need to translate the *reuse vector* in the iteration space to the data-footprint of the array for the iteration slice defined by the *reuse vector*. 

---

45
Perfectly Nested Loops

Consider a perfectly nested loop-nest (i.e., all the statements in the loop have the same loops surrounding them) with \( n \) loops having loop-iterators \(- i_1, i_2, \ldots, i_n \). Let the iteration space be represented as the following set \( I \).

\[
I = \{[i_1, \ldots, i_n] : c_1 \land \cdots \land c_n \}
\]

The loop-bounds of each of the loop-iterators \( i_1, \ldots, i_n \) are encoded in the constraints \( c_1, \ldots, c_n \) respectively.

Suppose that there exists a reuse vector \( \vec{d} = (d_1, d_2, \ldots, d_n) \) for an array \( X \) and the different references to \( X \) are given by the relations \( r_1, r_2, \ldots, r_m \).

We note that at least one of \( d_1, \ldots, d_n \) is not zero because, otherwise \( \vec{d} = \vec{0} \), in which case it represents the trivial reuse. We develop a solution to the estimation of working set size when only one of \( d_1, \ldots, d_n \) is not zero first and then construct the solution when any number of \( d_1, \ldots, d_n \) are not zero.

Let \( \vec{d}_k = (0, \ldots, d_k, \ldots, 0) \) for some \( k \). The iteration space with the structure and the number of iterations described by the reuse vector \( \vec{d}_k \) (i.e., iterations between a use and a reuse of data) may be modeled as the following set.

\[
I_{\vec{d}_k} = \{[i_1, \ldots, i_k, \ldots, i_n] : (1 \leq i_1 \leq 1) \land \cdots \land (1 \leq i_{k-1} \leq 1) \land (1 \leq i_k \leq |d_k|) \land c_{k+1} \cdots \land c_n \}
\]
The constraints \(c_{k+1} \ldots c_n\) are drawn from the original iteration space set \(I\) with a modification: variables \(i_1\) to \(i_{k-1}\), if appear in them, are projected to their respective lexicographic minimum or maximum so that the constraints result in the largest iteration space possible for any value of \(i_1\) through \(i_{k-1}\).

The \(i_k\) loop is set to execute \(|d_k|\) number of times and the loops inner to \(i_k\) are run through their entire lengths. The dimensions (loop-lengths) of the loops outer to \(i_k\) are set to 1 so that the execution reaches inner loops starting from \(i_k\) only once. Thus, by setting their lower and upper bounds to 1, they are effectively removed from consideration towards the calculation of working set size.

When the \(relations r_1, r_2, \ldots r_m\) are applied on the restricted iteration space \(I_{\tilde{d}_k}\), it gives the sets of elements of array \(X\) accessed by each reference \(r_i\) in the iteration space \(I_{\tilde{d}_k}\). The union of these sets constructs all the elements of \(X\) accessed collectively by the different references. Thus, the working set of the reuse vector \(\tilde{d}_k\) for the array \(X\) is computed as:

\[
WS_{\tilde{d}_k,X} = \{r_1(I_{\tilde{d}_k}) \cup r_2(I_{\tilde{d}_k}) \cup \cdots \cup r_m(I_{\tilde{d}_k})\}
\]

The working set size is the cardinality of the set \(WS_{\tilde{d}_k,X} = |WS_{\tilde{d}_k,X}|\).

The working set size for the general reuse vector \(\tilde{d}\) is calculated by decomposing \(\tilde{d}\) into vectors that have only one non-zero component and then summing working-set sizes of these component vectors.

\[
|WS_{\tilde{d},X}| = \sum_{d_k \neq 0} sign(d_k) \times |WS_{\tilde{d}_k,X}|
\]

**Example** An example of a perfectly nested loop is shown in Fig. 3.4. The array reference \(X[i_1][i_2]\) in the example has self-temporal reuse vector \(\tilde{d}_p = (1, 0, 0)\). That is, a particular
array element $X[i_1,p][i_2,p]$ accessed at the iteration $(t_p, i_1,p, i_2,p)$ will be accessed again at iteration $(t_p + 1, i_1,p, i_2,p)$. $\vec{d}_p$ has a non-zero entry in the first position and that is the only non-zero entry as well. Hence, the working set size for $\vec{d}_p$ is equal to the working set size for $\vec{d}_1$ (vector with non-zero entry present only in the first position). The restricted iteration space for $\vec{d}_1$ is given by the set:

$$I_{\vec{d}_1} = \{[t,i_1,i_2] : (1 \leq t \leq 1) \land (0 \leq i_1 < n) \land (1 \leq i_2 < n)\}$$

Two references to the array $X - X[i_1][i_2]$ and $X[i_1][i_2 - 1]$ are represented by relations:

$$r_1 = \{[t,i_1,i_2] \mapsto [i_1',i_2'] : (i_1' = i_1) \land (i_2' = i_2)\}$$
$$r_2 = \{[t,i_1,i_2] \mapsto [i_1',i_2'] : (i_1' = i_1) \land (i_2' = i_2 - 1)\}$$

The working set is the union of the sets $r_1(I_{\vec{d}_1})$ and $r_2(I_{\vec{d}_1})$.

$$WS_{\vec{d}_1,X} = \{r_1(I_{\vec{d}_1}) \cup r_2(I_{\vec{d}_1})\}$$

The working set size $= |WS_{\vec{d}_1,X}| = n^2$.

**Imperfectly Nested loops - Reuse Within a Loop**

In the code shown in Fig. 3.5, the two statements $S_1$ and $S_2$ have a common outer-most loop with iterator $t$ but have different inner loops. Therefore, such loops are imperfectly nested in the sense that all the statements in the loop-nest do not have the same loops surrounding them.
for (t = 0; t < steps; t++)
{
    for (i1 = 0; i1 < n; i1++)
        for (i2 = 1; i2 < n; i2++)
            S1: Access X[i1][i2], X[i1][i2−1];
    for (i1 = 0; i1 < n−2; i1++)
        for (i2 = 0; i2 < n; i2++)
            S2: Access X[n+i1−2][i2], X[n+i1−3][i2];
}

Figure 3.5: An imperfectly nested loop

Further, the array reference X[i1][i2] in the statement S1 has self-temporal reuse vector \( \vec{d}_p = (1, 0, 0) \): reuse of a particular element - \( X[i1_p][i2_p] \) - occurs in the next iteration of the \( t \) loop. This implies that the statement S2 would also be executed before the reuse of the element \( X[i1_p][i2_p] \). Since the cache is partitioned at the level of arrays (described in §5.7) and there are references to the same array \( X \) in the statement S2 also, if the cache cannot hold all the accesses to the array \( X \) from both the statements - S1 and S2 - between the accesses to \( X[i1_p][i2_p] \), then the reuse may not be realizable in cache. Therefore, the working-set size computation for a reuse of \( X[i1][i2] \) in the statement S1 needs to include references to the array \( X \) from both S1 and S2.

In general, for a given reuse vector \( \vec{d} = (d_1, d_2, \ldots, d_n) \), if the first non-zero entry from the beginning is \( d_j \) for some \( j \) i.e., all \( d_i \) for \( i < j \) are zero, then we say that reuse is at the level \( j \): the reuse occurs after \( d_j \) iterations of the corresponding loop \( i_j \). Hence, the working-set calculation considers all the statements enclosed within the loop \( i_j \).

Returning to the example considered before - the one corresponding to the code in Fig. 3.5 -, working-set size estimation for the reuse vector \( \vec{d}_p = (1, 0, 0) \) proceeds as follows.

The iteration spaces of the statements S1 and S2 having the number of iterations specified in \( \vec{d}_p \) are modeled as the sets \( I_{S1} \) and \( I_{S2} \):

49
\[ I_{S1} = \{ [t, i1, i2] : (1 \leq t \leq 1) \land (0 \leq i1 < n) \land (1 \leq i2 < n) \} \]

\[ I_{S2} = \{ [t, i1, i2] : (1 \leq t \leq 1) \land (0 \leq i1 < n-2) \land (0 \leq i2 < n) \} \]

Four references to the array \( X[i1][i2], X[i1][i2-1], X[n+i1-2][i2], X[n+i1-3][i2] \) - are characterized by the relations:

\[ r_1 = \{ [t, i1, i2] \mapsto [i1', i2'] : (i1' = i1) \land (i2' = i2) \} \]

\[ r_2 = \{ [t, i1, i2] \mapsto [i1', i2'] : (i1' = i1) \land (i2' = i2 - 1) \} \]

\[ r_3 = \{ [t, i1, i2] \mapsto [i1', i2'] : (i1' = n+i1-2) \land (i2' = i2) \} \]

\[ r_4 = \{ [t, i1, i2] \mapsto [i1', i2'] : (i1' = n+i1-3) \land (i2' = i2) \} \]

The working set is the union of the sets \( r_1(I_{S1}), r_2(I_{S1}), r_3(I_{S2}) \) and \( r_4(I_{S2}) \).

\[ WS_{\tilde{d}_p,X} = \{ r_1(I_{S1}) \cup r_2(I_{S1}) \cup r_3(I_{S2}) \cup r_4(I_{S2}) \} \]

The working set size = \( |WS_{\tilde{d}_p,X}| = 2 \times n^2 - 5 \times n + 4 \).

**Imperfectly Nested loops - Reuse Between Loops**

In addition to the reuse within a loop as seen hitherto, reuse of data between loops - data brought by statements in an earlier loop-nest being used by statements in a later loop-nest - is also possible.

As an example, consider the code shown in Fig. 3.6. Statement \( S1 \) in the first loop-nest accesses rows 0 to \( n - 1 \) of array \( X \) in that order. Statement \( S2 \) of the second loop-nest touches rows \( n - 2 \) to 0 of the same array successively. Therefore, if two rows - \( n - 1 \) and \( n - 2 \) - of \( X \) brought in by \( S1 \) are retained in cache, then elements of \( (n-2)^{th} \) row can be
Figure 3.6: An imperfectly nested loop with reuse between loop-nests

reused by $S_2$. Similarly, if all the $n$ rows can be fitted in cache, reuse of $n - 1$ rows - 0 to $n - 2$ - is feasible.

In this section, a procedure to automatically detect reuse between loops and to quantify the working set size is developed.

Let an array be accessed in two adjoining loop-nests $LN_1$ and $LN_2$ ($LN_1$ appears before $LN_2$). The iteration spaces of $LN_1$ and $LN_2$ are modeled by sets $I_1$ and $I_2$. The array access functions in $LN_1$ are represented by relations $r_{1,1}, \ldots, r_{m,1}$ and those in $LN_2$ by relations $r_{2,1}, \ldots, r_{n,1}$.

The data foot-prints of the array in $LN_1$ and $LN_2$ are calculated to be $D_1$ and $D_2$ respectively.

$$D_1 = \{r_{1,1}(I_1) \cup \cdots \cup r_{1,m}(I_1)\}$$

$$D_2 = \{r_{2,1}(I_2) \cup \cdots \cup r_{2,n}(I_2)\}$$

The common data used between the loops is the intersection of the two data foot-prints $D_1$ and $D_2$: $D_{\text{common}} = \{D_1 \cap D_2\}$

If $D_{\text{common}}$ is non-empty, then there exists reuse between the two loops and the analysis advances to working set size computation.
The working-set in this case is the union of elements of the loops LN1 and LN2 that are required to be held in cache in order that data brought by LN1 - D_common - is reused in LN2.

The iterations of LN1 that bring in elements of D_common into cache for the last time during the execution of LN1 - LN1_last - and the iterations of LN2 that use the elements of D_common for the very first time during the execution of LN2 - LN2_first - are characterized.

The minimum working-set - W_min - needed to exploit reuse of at least one element comprises of data accessed from the last iteration of LN1, last to the first iteration of LN2, first.

The maximum working-set - W_max - required to exploit reuse of all the elements of D_common consists of data accessed from the first iteration of LN1, last to the last iteration of LN2, first. More formally,

\[ \forall \text{elmt} \in D_{\text{common}} : \]

\[ \text{LN}_{1, \text{elmt}, \text{last}} = \text{lexmax}\{ (r_{1,1}^{-1}(\text{elmt}) \cap I_1) \cup \cdots \cup (r_{1,m}^{-1}(\text{elmt}) \cap I_1) \} \]

\[ \text{LN}_{2, \text{elmt}, \text{first}} = \text{lexmin}\{ (r_{2,1}^{-1}(\text{elmt}) \cap I_2) \cup \cdots \cup (r_{2,n}^{-1}(\text{elmt}) \cap I_2) \} \]

\[ \bar{d}_{1, \text{elmt}} = \text{lexmax}\{ I_1 \} - \text{LN}_{1, \text{elmt}, \text{last}} \]

\[ \bar{d}_{2, \text{elmt}} = \text{LN}_{2, \text{elmt}, \text{first}} - \text{lexmin}\{ I_2 \} \]

\[ W_{\bar{d}_{1, \text{elmt}}} = \{ r_{1,1}(I_{\bar{d}_{1, \text{elmt}}}) \cup \cdots \cup r_{1,m}(I_{\bar{d}_{1, \text{elmt}}}) \} \]

\[ W_{\bar{d}_{2, \text{elmt}}} = \{ r_{2,1}(I_{\bar{d}_{2, \text{elmt}}}) \cup \cdots \cup r_{2,n}(I_{\bar{d}_{2, \text{elmt}}}) \} \]

\[ W_{\text{elmt}} = \{ W_{\bar{d}_{1, \text{elmt}}} \cup W_{\bar{d}_{2, \text{elmt}}} \} \]

The minimum working-set is the working-set of that element whose working-set size the smallest over all the elements in D_common. The maximum working-set is the union of working-sets of all the elements in D_common.
\[
WS_{\text{min}} = WS_x : |WS_x| = \min_{\text{elmt} \in D_{\text{common}}} |WS_{\text{elmt}}|
\]
\[
WS_{\text{max}} = \bigcup_{\text{elmt} \in D_{\text{common}}} WS_{\text{elmt}}
\]

Thus, determination of \(WS_{\text{min}}, WS_{\text{max}}\) entails working-set calculations ‘for all’ elements in \(D_{\text{common}}\). This poses a computational challenge for the general case when the sets are parametric and hence, elements in the set cannot be explicitly enumerated.

However, if all the array references in the loops \(LN_1\) and \(LN_2\) are either lexicographically increasing or decreasing (individually, need not be across the loops) then the lexicographic minimum and maximum of the set \(D_{\text{common}}\) may be used as proxies for the whole set. This is because working-set of one of the extrema would be \(WS_{\text{min}}\) and working-set of the other extremum would be \(WS_{\text{max}}\) as working-set of any element belonging to \(D_{\text{common}}\) would be a subset of working-set of this extremum.

If the aforementioned condition is not met, then a lower bound and an upper bound on the working-set size are derived using optimistic and pessimistic assumptions: To derive the lower bound, we assume that between the use of \(D_{\text{common}}\) in \(LN_1\) and \(LN_2\), there are no other intervening accesses to any other data elements which do not belong to \(D_{\text{common}}\). While deriving the upper bound, we assume that the data elements brought in by \(LN_1\) which are not reused in \(LN_2\) are brought-in during the end iterations of \(LN_1\) and the data that is not common between \(LN_1\) and \(LN_2\) is used during the initial iterations of \(LN_2\). This assumption provides the largest possible working-set.

\[
WS_{\text{min}} = \{D_{\text{common}}\}
\]
\[ WS_{\text{max}} = \{(D_1 \setminus D_{\text{common}}) \cup (D_2 \setminus D_{\text{common}}) \cup D_{\text{common}}\} = \{D_1 \cup D_2\} \]

**Example**  
Reuse exists between loops shown in Fig. 3.6 as discussed earlier.

Iteration spaces of the statements \( S_1 \) and \( S_2 \) are represented using the sets \( I_{S_1} \) and \( I_{S_2} \) mentioned in §3.4.1. Iteration space of the first loop in the example \( LN_1 \) is the set \( I_{S_1} \) and that of the second loop \( LN_2 \) is \( I_{S_2} \). The array references in \( S_1 \) are translated to relations \( r_1 \) and \( r_2 \) as indicated in the same section §3.4.1.

The array references to the array \( X \) in \( S_2 - X[n - i_1 - 2][i_2], X[n - i_1 - 3][i_2] \) - are characterized using the following relations:

\[
\begin{align*}
    r_5 &= \{(t, i_1, i_2) \mapsto [i_1', i_2'] : (i_1' = n - i_1 - 2) \land (i_2' = i_2)\} \\
    r_6 &= \{(t, i_1, i_2) \mapsto [i_1', i_2'] : (i_1' = n - i_1 - 3) \land (i_2' = i_2)\}
\end{align*}
\]

The data foot-print of the array \( X \) in \( LN_1 \) is \( D_1 \) and the same in \( LN_2 \) is \( D_2 \) which are computed as the following unions of sets. \( D_{\text{common}} \) is the intersection of the sets \( D_1 \) and \( D_2 \).

\[
\begin{align*}
    D_1 &= \{r_1(I_{S_1}) \cup r_2(I_{S_1})\}, \\
    D_2 &= \{r_5(I_{S_2}) \cup r_6(I_{S_2})\} \\
    D_{\text{common}} &= \{D_1 \cap D_2\}
\end{align*}
\]

The references \( r_1 \) and \( r_2 \) appearing in \( LN_1 \) are both lexicographically increasing. However, the references \( r_5 \) and \( r_6 \) of \( LN_2 \) are not strictly lexicographically increasing or decreasing. For example, the accesses due to \( r_5 \) are: \( X[n - 2][0], X[n - 2][1], \ldots, X[n - 3][0], \ldots \)

Thus, the elements are accessed in descending row order but are referenced in ascending order within a row. The analysis therefore falls back to conservative estimations.
\[ |WS_{\text{min}}| = |D_{\text{common}}| = n^2 - n \]
\[ |WS_{\text{max}}| = |D_1 \cup D_2| = n^2 \]

3.4.2 NWSL: Number of Working Set Cache Lines

The number of cache-lines required to accommodate a given working-set in cache is influenced both by its size and the data layout of the working-set elements in memory.

If \( L \) elements can fit in a line of cache and all of the working-set elements are consecutive in memory then the number of cache-lines needed to fit a working-set of size \( N \) would be approximately \( \lceil \frac{N}{L} \rceil \). On the other hand, if the elements are scattered in memory and no two of them map to the same cache-line then \( N \) cache-lines are needed. In some other scenario, only a subset of the elements may be consecutive in memory, in which case we would need more than \( \lceil \frac{N}{L} \rceil \) cache-lines but less than \( N \).

To estimate the number of cache-lines necessary to fit a working-set, the linear addresses of the array elements in the working-set are constructed assuming that the first element of the array is located at address 0. Then, \( L \) consecutive linear addresses are mapped to the same cache-line. The cardinality of the set of cache-lines formed from the working-set provides the estimation for the number of cache-lines needed.

For illustration, we consider the code example shown in Fig. 3.4 whose working-set was calculated in §3.4.1.

Assuming, row-major ordering of the array elements in memory; \( n = 1000 \); size of the array is 2000 in each of its two dimensions; and each array element is 8 bytes long, the linear addresses from access functions are computed using the following relation.
Supposing that the cache-line is 64 bytes long, linear-address to cache-line mapping is formulated as:

\[
\mathbf{r}_{\text{linear}} = \{[i_1, i_2] \mapsto [2000 \times 8 \times i_1 + 8 \times i_2]\}
\]

Thus, the set of cache-lines is defined as:

\[
\mathbf{r}_{\text{line}} = \{[i] \mapsto [i'] : i' = \lfloor i/64 \rfloor\}
\]

The number of cache-lines needed is the cardinality of the above set: \(|\mathbf{W}_{\Sigma L_{d_1, X}}| = 125000\).

For the general reuse vector \(\vec{d} = (d_1, d_2, \ldots, d_n)\), the number of working set lines (\(\text{NWSL}\)) is calculated by the following summation.

\[
|\mathbf{W}_{\Sigma L_{\vec{d}, X}}| = \sum_{d_k \neq 0} \text{sign}(d_k) \times |\mathbf{W}_{\Sigma L_{d_k, X}}|
\]

where, \(\mathbf{W}_{\Sigma L_{d_k, X}} = \{\mathbf{r}_{\text{line}}(\mathbf{r}_{\text{linear}}(\mathbf{W}_{\Sigma L_{d_k, X}}))\}\) and \(\mathbf{r}_{\text{line}}, \mathbf{r}_{\text{linear}}\) are defined taking cache structure and array dimensions into account.

### 3.4.3 Useful Cache Size Calculation Algorithm

**Algorithm 1 Compute_NWSL**

**Input:** Array: \(\text{Arr}\), Reuse-Vector: \(\text{ReuseVector}\), Affine Loop-nest: \(P\), Problem Sizes: \(N[\#\text{Arrays}]\).

**Output:** Number of Working Set Cache Lines: \(\text{NWSL}\).

1. ReuseIterationSlice ← Form an iteration slice in \(P\) corresponding to \(\text{ReuseVector}\)
2. \(\text{WS} ← \text{Compute data footprint of references of } \text{Arr} \text{ in } \text{ReuseIterationSlice}\)
3. \(\text{NWSL} ← \text{Compute number of cache lines for } \text{WS} \text{ in the data layout defined by problem sizes } N\)

return \(\text{NWSL}\)
Algorithm 2 Calculate_Useful_Cache_Size

**Input:** Affine Loop-nest: $P$, Problem Sizes: $N[\#\text{Arrays}]$, Maximum number of Cache-lines: $MaxCacheLines$

**Output:** Useful number of Cache-lines: $UsefulCacheLines$, PerArrayAllocation: $PerArrayCacheLines$

- $WorkingSetLines \leftarrow \phi$
- for all Array $Arr$ in $P$ do
  - $ReuseVectors \leftarrow \text{PolyhedralDependencies}(P)$
  - for all $ReuseVector$ in $ReuseVectors$ do
    - $NumWorkingSetLines \leftarrow \text{Compute_NWSL}(Arr, ReuseVector, P, N)$
    - $\text{AddToWorkingSetLines}(Arr, NumWorkingSetLines)$
  - end for
- end for

- $\text{SortWorkingSetLinesInIncreasingOrder}(WorkingSetLines)$
- Initialize $PerArrayCacheLines[\#\text{Arrays}] \leftarrow 0$
- $UsefulCacheLines \leftarrow 0$
- for all $NumWorkingSetLines$ in $WorkingSetLines$ do
  - $Arr \leftarrow \text{GetCorrespondingArray}(NumWorkingSetLines)$
  - if $NumWorkingSetLines \leq (MaxCacheLines - UsefullCacheLines + PerArrayCacheLines(Arr))$ then
    - $UsefulCacheLines \leftarrow (UsefulCacheLines - PerArrayCacheLines(Arr)) + \text{NumWorkingSetLines}$
    - $PerArrayCacheLines(Arr) \leftarrow \text{NumWorkingSetLines}$
  - end if
  - end for
- return $UsefulCacheLines, PerArrayCacheLines$

Algorithm 1 puts together the steps for computing working sets described in §3.4.1, §3.4.2. The general iteration slice of a given loop nest that models iterations between the source and target of a reuse vector is formed. This step formulates iteration slice taking cognizance of the nature of reuse: if reuse is in a perfectly/imperfectly nested loop; whether it is within a loop or across loops. Then, data footprint of the references of the array in the iteration slice is determined. For the working set elements thus computed, the number of cache lines they map to, are estimated considering data layout of the array elements in memory.

Algorithm 2 is used to calculate the useful cache size for a given affine program and system configuration. Input to the algorithm are the affine loop-nest (a sequence of loop-nests if reuse between them is detected), the actual problem sizes (and tile sizes) used and the maximum available cache size. Polyhedral dependence analysis [14] is invoked to obtain reuse distance vectors. Then, for each reuse vector, the number of working set cache
lines required to exploit reuse corresponding that reuse vector is computed using Algorithm 1.

The working-set cache lines determined are arranged in increasing order to derive per-array allocations of the available cache lines. The available cache lines are apportioned to reuse-vectors in increasing order of their number of working set lines. This precludes the possibility of one array exhausting the cache while other arrays have reuses with smaller working-sets.

The amount of cache that is earmarked thus is output as the “useful cache”. The number of cache lines to be assigned to each array are also specified.

### 3.5 Experimental Evaluation

In this section, we describe experiments conducted to first assess the accuracy of the compiler analysis and then use it to perform a characterization of the useful cache size and potential power savings for a number of benchmarks over a range of problem sizes.

To implement the per-array cache allocations indicated by the compiler analysis, we assume a page-coloring [114, 75] mechanism to partition cache among different variables in the program. It works as follows. A subset of set index bits of a physical address corresponds to the physical page number. These bits are referred to as the “page color” of the physical address. If the cache is to be partitioned amongst certain variables, it can be achieved by mapping those variables to different pages (and hence assigning them different page colors). This enables different arrays to be mapped to different sets in cache. Additionally, the fraction of cache allocated to a particular array can also be controlled by placing the array in pages possessing a certain number of colors.
As a case study, Intel Core i7-965 Extreme Edition machine is used and compiler analysis is performed on the benchmark programs to determine useful-cache-size based on cache parameters of the machine. For each benchmark, the difference in cache miss-rates with the original cache and the resized cache is recorded. The power calculations (based on CACTI [18]) are made to estimate the reduction in leakage power with the resized cache.

Table 3.1: Cache configuration of Intel Core i7-965 Extreme Edition

<table>
<thead>
<tr>
<th>Cache</th>
<th>Size</th>
<th>Associativity</th>
<th>Line-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private L1 cache</td>
<td>32KB</td>
<td>8-way</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Private L2 cache</td>
<td>256KB</td>
<td>8-way</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Shared L3 cache</td>
<td>8MB</td>
<td>16-way</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Per-core share of L3 cache</td>
<td>2MB</td>
<td>16-way</td>
<td>64 bytes</td>
</tr>
</tbody>
</table>

The cache configuration of the said Core i7 machine is shown in Table 3.1. The system has 4 cores and 8MB of L3 cache shared among them. For experimental evaluations, since we model sequential programs in this work, each core is assumed to have a 2MB L3 cache.

3.5.1 Benchmarks

Nine benchmarks programs - Seidel, tiled Seidel, STREAM Triad, ADI, gemm, tiled gemm, bicg, Jacobi-2D and tiled Jacobi-2D - are used for the experiments. Seidel is a stencil that is used in the Gauss-Seidel method for iteratively solving a linear system of equations. STREAM Triad [79] is designed to measure sustainable memory bandwidth. We use sequential version of it here as a representative for streaming applications. ADI is a stencil used in solving partial differential equations (PDE’s). gemm is a linear algebra kernel that computes $C := \alpha A B + \beta C$. bicg implements the biconjugate gradient method.
Jacobi-2D is a stencil code that is used in Jacobi method to compute solutions for systems of linear equations. Seidel, gemm, ADI, bicg, Jacobi-2D source codes are extracted from PolyBench, a polyhedral benchmark suite [94].

Seidel, gemm and Jacobi-2D codes are tilable, hence both un-tiled and tiled versions of them are used. The compiler-analysis developed in this chapter assumes that tile sizes are given. Seidel, gemm and Jacobi-2D benchmarks have loops of depth 3 and hence, in the tiled codes of each of the benchmarks there are 3 tile sizes to be chosen. For the purpose of experiments, tile sizes are selected using auto-tuning as follows. The cache miss-rates are obtained with the cache simulator - Dinero [34] - for square tile sizes (the 3 tile sizes are set equal) between (16, 16, 16) and (512, 512, 512) and the memory times are estimated for Core-i7 machine. The memory time estimations are based on the latencies of L1 cache, L2 cache, L3 cache and main memory in the Core-i7 machine, which are, 4, 11, 39 and 107 cycles respectively. The tile sizes which result in minimum memory time are determined and the useful-cache-size analysis is performed using those tile sizes.

ADI is not time-tilable since its outermost loop cannot be permuted with other loops. Therefore, un-tiled ADI is employed for the experiments. In bicg, for the problem sizes considered, all the working sets fit in L3 cache. Therefore, experiments are conducted only with un-tiled bicg.

The useful-cache-size calculation algorithm is implemented using iscc [61], a library for counting the number of integer points in parametric and non-parametric polytopes. Using the per-array cache allocations and the total useful cache size suggested by the algorithm, virtual addresses corresponding to data accesses by the benchmark programs are page-colored and passed through the cache simulator - Dinero [34]. The miss rates output by it are noted.
The power measurements are made using CACTI [122, 18]. The experimental set-up is depicted in Fig. 3.7.

### 3.5.2 Results

#### Validation of Analysis Accuracy

Table 3.2: Cache miss-rates and leakage-power in Original and Resized L3 Cache

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem Size</th>
<th>Original L3 Cache</th>
<th>Resized L3 Cache</th>
<th>Power Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Miss-rate</td>
<td>Leakage Power (in mW)</td>
<td>Cache Size</td>
</tr>
<tr>
<td>Seidel</td>
<td>200X800</td>
<td>0.0126</td>
<td>6877.43</td>
<td>32KB</td>
</tr>
<tr>
<td>Tiled Seidel</td>
<td>200X800</td>
<td>0.0004</td>
<td>6877.43</td>
<td>1MB</td>
</tr>
<tr>
<td>STREAM Triad</td>
<td>800</td>
<td>0.1250</td>
<td>6877.43</td>
<td>4KB</td>
</tr>
<tr>
<td>ADI</td>
<td>200X800</td>
<td>0.0467</td>
<td>6877.43</td>
<td>2MB</td>
</tr>
<tr>
<td>gemm</td>
<td>800</td>
<td>0.0313</td>
<td>6877.43</td>
<td>256KB</td>
</tr>
<tr>
<td>Tiled gemm</td>
<td>800</td>
<td>0.0011</td>
<td>6877.43</td>
<td>1MB</td>
</tr>
<tr>
<td>bicg</td>
<td>800</td>
<td>0.0251</td>
<td>6877.43</td>
<td>32KB</td>
</tr>
<tr>
<td>Jacobi-2D</td>
<td>200X800</td>
<td>0.0628</td>
<td>6877.43</td>
<td>64KB</td>
</tr>
<tr>
<td>Tiled Jacobi-2D</td>
<td>200X800</td>
<td>0.0011</td>
<td>6877.43</td>
<td>2MB</td>
</tr>
</tbody>
</table>
Table 3.2 lists the benchmarks and the problem sizes used in the experiments. The cache is resized to the smallest power-of-two size according to the compiler-analysis-determined useful number of cache lines. Analytically determined cache sizes are shown under the column ‘Cache Size’. The miss rates with the original 2MB cache and the resized cache are also indicated in the table. The cache miss rates obtained by Dinero simulation remain unchanged for all the benchmarks when the size of the L3 cache is resized based on the analysis of useful cache size.

The leakage power measurements for the original and resized cache are presented in the same table (Table 3.2). The original L3 cache dissipates 6877.43 mW of leakage power. The streaming benchmark - STREAM Triad - records the highest 98.73% power savings with its resized cache capacity of 4KB. It has no self-temporal reuse or group-reuse but only has self-spatial reuse. Hence, the working set is small for STREAM Triad. Seidel, bicg, Jacobi-2D are the other benchmarks that register over 90% in leakage power reductions. In the case of ADI and Tiled Jacobi-2D, the whole cache capacity is deemed useful.

We note that for the shared 8MB L3 cache in the Intel Core i7 processor, the power consumption would be nearly 27.52W and is 21.17% of the max TDP of 130W [29]. Thus, the leakage power of the L3 cache constitutes a sizable portion of the power budget of the processor.

The validation experiments were conducted for 5 other problem sizes for each of the 9 benchmarks: cache miss-rates were obtained for problem sizes 200, 1500, 2000, 2500, 3000 with the full cache and analysis-determined resized cache. In 42 out of the 45 cases, the compiler-analysis was exact in that, there was no miss-rate increase with the resized cache; in 3 other instances, there was a miss-rate increase due to conflict misses. For tiled Jacobi-2D, at problem sizes 2500 and 3000, there were miss-rate increases of 0.0025 and
0.0026 respectively. A 0.0001 miss-rate increase was observed in ADI for problem size 200.

**Variation of Useful Cache-size with Problem Size**

To study how the useful-cache-size changes for different problem sizes, compiler-analysis is invoked for problem sizes ranging from \( N = 10 \) to \( N = 15000 \) in steps of 10. For stencils, the number of time steps is set to 200. Figures 3.8 to 3.16 plot the percentage of L3 cache that can be turned off without affecting performance as the problem sizes are varied for various benchmarks.

For STREAM Triad, the amount of L3 cache that can be turned-off stays at 98.73\% irrespective of the problem size (Fig. 3.11). Fig. 3.14 plots the cache size reduction for \texttt{bicg}. The \texttt{bicg} code uses 5 arrays and the working set size of any array to realize all the reuses is no greater than \( N \) elements each. Thus, even for the maximum problem size...
Figure 3.9: Tiled Seidel: Percentage of Non-useful Cache Size

Figure 3.10: ADI: Percentage of Non-useful Cache Size
Figure 3.11: STREAM Triad: Percentage of Non-useful Cache Size

Figure 3.12: gemm: Percentage of Non-useful Cache Size
Figure 3.13: Tiled gemm: Percentage of Non-useful Cache Size

Figure 3.14: bicg: Percentage of Non-useful Cache Size
Figure 3.15: Jacobi-2D: Percentage of Non-useful Cache Size

Figure 3.16: Tiled Jacobi-2D: Percentage of Non-useful Cache Size
considered - 15000 -, all working sets fit in cache. Hence, the extent of cache that can be turned off steadily decreases as the problem size is increased.

The graphs of other benchmarks (Seidel, Tiled Seidel, ADI, gemm, Tiled gemm, Jacobi-2D) indicate that initially when the problem size is small and is much less than L3 cache size, the remainder of cache can be turned off. However when the problem size is increased, the difference between the cache size and the problem size decreases and the curve tapers off to zero. As problem size is further increased and the whole problem (all the data used by the program) can no longer fit in L3 cache, the next smaller working-set that can fit into cache is much smaller than the L3 cache size. Therefore much of the L3 cache may be turned off again. For example, in Seidel (Fig. 3.8), when the whole problem ($N^2$ elements) does not fit in cache, the next reuse occurs at a working set of size $3N$ elements. Therefore, starting from $N = 520$, a large fraction of cache may be turned off; the fraction decreases as $N$ is increased.

The tiled codes have twice the number of loops as their un-tiled counterparts and as a result, have a larger number of reuse vectors and possible working-sets. Consequently, we see more transition points in the case of tiled codes - tiled Seidel (Fig. 3.9), tiled gemm (Fig. 3.13), tiled Jacobi-2D (Fig. 3.16). Once the problem size is sufficiently large, the only working set that can fit in cache is the footprint of a tile (which is a function of tile sizes alone and not problem size) and in these scenarios, excess of L3 cache beyond that is needed to hold the working-set of a tile may be power-gated.

Fig. 3.10 presents the non-useful-cache percentages against problem sizes for ADI. When the problem size fits wholly into L3 cache, the remainder of cache can be turned off. Once the problem size exceeds L3, there always exist working-sets smaller than L3
Table 3.3: Percentage of Reduced Cache across Problem Sizes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Reduced Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seidel</td>
<td>85.60%</td>
</tr>
<tr>
<td>Tiled Seidel</td>
<td>74.48%</td>
</tr>
<tr>
<td>STREAM Triad</td>
<td>99.80%</td>
</tr>
<tr>
<td>ADI</td>
<td>1.03%</td>
</tr>
<tr>
<td>gemm</td>
<td>43.77%</td>
</tr>
<tr>
<td>Tiled gemm</td>
<td>85.80%</td>
</tr>
<tr>
<td>bccg</td>
<td>79.13%</td>
</tr>
<tr>
<td>Jacobi-2D</td>
<td>73.18%</td>
</tr>
<tr>
<td>Tiled Jacobi-2D</td>
<td>81.58%</td>
</tr>
</tbody>
</table>

cache but close to it that no non-useful cache can be found: working-sets of sizes $(N - 1) \times N, (N - 2) \times N, \ldots, 3N, 2N$.

Table 3.3 aggregates the percentage of cache that can be turned off across problem sizes for different benchmarks.

3.6 Related Work

We are unaware of any other work that has proposed a static compile-time approach to estimating the useful fraction of a cache based on reuse analysis. We discuss below related work from the architectural side on cache customization and also previously proposed compile-time reuse analysis techniques.

3.6.1 Cache Customization

Grun et al. propose a double cache architecture containing a spatial cache and a temporal cache [51]. In their approach, the variables are statically allocated to one or the other cache based on access pattern with the overall objective of minimizing power consumption.
without impacting performance. Hardavellas et al. characterize the data accesses into 3 classes: Private data, Instructions and Shared data and then place data in distributed caches to optimize for low latency access [53].

The drowsy caches [37] are proposed to save leakage power by putting some cache lines into a state-preserving low power drowsy mode. The ways are incrementally activated only when the cache miss rate increases. Molecular Caches [117] are custom-built for applications running on CMPs, which are aggregates of different chunks of the overall cache present on the chip. Molecular caches are shrunk or expanded depending on the need of the application for power savings.

Srikantaiah et al. develop a reconfigurable adaptive multi-level cache hierarchy. By using an active cache footprint estimation hardware technique, they improve performance of multi-threaded and multi-programmed workloads [110]. Jiang et al. develop asymmetric caches (small and big caches) to satisfy varying application requirements so that chip area and energy may be saved [63].

An off-line Shared cache modeling technique to estimate the working sets of sequential programs when running in parallel is developed by Xiang et al. [126]. In a hybrid software and hardware approach [49], at compile time the memory accesses are classified into high-locality and irregular categories and are steered towards the caches optimized for the respective access patterns. Lopez et al. present resizable caches based on cache hit behavior [74].

Zhang et al. develop a compiler approach to reduce leakage energy by activating and deactivating particular cache lines based on reuse of data [132]. A compiler-pass instruments the program with activate and deactivate instructions and hardware capability to
turn on and off individual cache lines is assumed to implement the instructions. The approach however increases the execution time of applications due to overheads associated with activation and deactivation of cache lines.

### 3.6.2 Reuse Analysis

Many studies have addressed cache reuse analysis in the context of estimating cache misses for performance prediction and in the selection optimal tile sizes for tiled code. These works are presently discussed.

Cascaval et.al [21, 20] develop a compile-time algorithm to build stack distance histogram for affine programs using data dependence distance vectors. The iteration space is partitioned such that in each partition, each instance of an array reference has the same incoming dependences. The number of distinct elements accessed in the dependence spans in each partition gives the stack distances for array references in those partitions. The method developed is applicable for analysis of reuse within a loop but analysis of reuse across loops is not handled.

Ghosh et.al introduce a framework for *Cache Miss Equations* (CME’s) to calculate the number of cache misses in affine programs [44]. For each reuse vector, two kinds of equations are generated: *compulsory equations* which represent cold misses and *replacement equations* which represent conflict and capacity misses. The number of cache misses is computed by traversing the iteration space and computing solutions to cache misses at each iteration point (the system is not parametric). Xue and Vera [118, 128] generalize reuse vectors to capture reuse between multiple loop nests; the call statements are abstractly inlined. This work uses the CME approach [44] to specify cache misses using mathematical formulas. Statistical sampling techniques are used to derive *miss ratio* of applications from these
formulas. While the approach could be used indirectly to determine useful cache capacity by repeatedly solving the CME system for different cache sizes, it would be prohibitively expensive to use in practice.

Chatterjee et.al [22] employ Presburger arithmetic to express cache misses. The cache misses are grouped into two classes: *interior misses* - guaranteed misses either due to capacity constraints or conflict in cache; *potential boundary misses* - those which may be misses/hits depending on the initial state of the cache. The techniques are general enough to determine cache misses in imperfectly nested loops but inter-loop reuse is not modeled. A significant disadvantage of the approach is the super-exponential computational complexity resulting from satisfiability checking and quantifier elimination of Presburger formulas.

Shirako et al. develop an analytical approach to bounding the range to search for optimal tile sizes [105] by combining a prior conservative model, DL [43], with a new optimistic model called ML. A lower and an upper bound are developed for the data footprint of a tile with symbolic tile sizes. Tile sizes are then chosen such that tile footprint fits in the cache of interest. The approach does not model inter-tile reuse and inter-loop reuse and is also only applicable for perfectly nested loops.

The working set procedure developed in this chapter improves over prior works in one or more of the following three aspects: i) accuracy in modeling reuse, ii) generality, and iii) practicality. The procedure considers self and group reuse, intra-loop and inter-loop reuse and is applicable to perfectly nested loops as well as imperfectly nested loops. Other than Vera et al.’s statistical sampling based approach [118], none of the prior works consider inter-loop reuse. The practicality of the algorithm stems from its formulation as well as its objective (i.e., to compute possible working sets) which is different from cache miss estimation. For example, a cache miss estimation algorithm has to consider initial iterations
which are not targets of dependence different from rest of the iterations; however, a working set procedure can ignore the initial iteration points as there is no data reuse in them and instead, can consider only the steady state iteration slice defined by a reuse vector to compute the corresponding working set.

3.7 Discussion and Conclusion

The sizes of the last level cache have been growing with each processor generation and they account for 40% or more of the die area on current machines. At the same time, leakage power constitutes 20-40% of the total power dissipation of the processor. In scenarios where the last level cache cannot be effectively used due to inherent data reuse characteristics of applications, certain fractions of it may be power-gated to save leakage power. In this chapter, we have developed a compile time approach to analyzing reuse characteristics of programs and determining useful cache size. Experimental evaluations demonstrate that the compiler analysis is effective in determining useful cache size and there is a significant potential to save power based on the analysis results.

A current limitation of the analysis technique developed in the chapter is that it models the behavior of fully associative caches. However, in the validation experiments conducted (§3.5.2) for 6 different problem sizes for 9 benchmark programs (a total of 54 combinations), in 51 cases, there was no error whatsoever in the useful cache size computed - Dinero simulation using the full cache and reduced L3 cache modeling the actual set associativity generated exactly the same number of misses. In the three instances where it did have an effect, a small change in miss-rate was observed.

The use of software page-coloring technique to partition cache among different arrays mitigates interference effects between arrays. Therefore, the only source of conflict
misses in cache arises from interference effects within individual arrays. The analysis approach developed in this chapter could also be applied for cache systems without assuming partitioning via software page-coloring. In this case, an extension of the approach using techniques from Smith’s probabilistic formulation [109] may be needed to maintain good accuracy.
Chapter 4: Compiler Support for Software Cache Coherence

4.1 Introduction

Continuous transistor scaling has enabled increasing amounts of logic and memory on a chip. The resulting higher computational density has been used to design highly-capable many-core compute accelerators. These platforms are programmed with highly-parallel applications, which are partitioned into many threads running in parallel on the different cores.

Unlike general-purpose parallel processors, these compute accelerators do not necessarily have hardware-coherent caches. For example, commercial Graphics Processing Units (GPUs) do not have full cache coherence [1, 2]. There are several reasons behind this lack of hardware cache coherence.

First, hardware cache coherence comes with a non-trivial implementation cost. It is difficult to design and verify cache coherence protocols completely [3]. Furthermore, coherence structures such as the directory storage, typically consume a considerable amount of chip area. In many compute accelerators, such chip area is best applied to increase the computational capabilities.
Secondly, the programs that run on accelerators have intrinsically less need for full hardware cache coherence. This is because these programs are usually written in epoch-based form. They have little data sharing among cores within an epoch, while most communication occurs across epoch boundaries. Consequently, the coherence requirements of the workloads can be largely enforced at epoch boundaries.

There has been significant interest in trying to understand how to best support cache coherence in accelerator architectures. At one extreme, GPU architectures do not have any hardware cache coherence. The Intel Single Chip Cloud processor [58, 77] does not provide hardware cache coherence, but instead defines a new type of memory to facilitate communication between cores. Other designs provide limited cache coherence for programmability, but with lower hardware overhead. For example, Runnemede from the DARPA UHPC program [19] provides scratchpads and software-managed incoherent caches, shifting the responsibility of coherence to the software. The Rigel accelerator architecture has support for a hybrid hardware-software cache coherence [67].

At the other extreme, the Intel Xeon Phi accelerator has fully coherent caches, largely motivated from keeping the task of programming as close as possible to existing models. Thus we have a range of current trends, because of the different demands and constraints. At one end, we have fully non-coherent caches in GPUs, which makes the hardware more easily scalable but forces a restrictive programming model and places a greater burden on application developers. At the other end, the Xeon Phi has sacrificed some hardware scalability in order to make the programming model easy for users by providing fully coherent caches. Software controlled caches could provide the best of both worlds if compilers could shoulder the burden: provide users the same epoch-based parallel programming model like
OpenMP that systems like the Xeon Phi provide, but allowing a simpler and more scalable hardware design like non-coherent GPUs.

In this chapter, we present some compiler advances that make software coherent systems more attractive. We divide problem domains into applications with regular memory accesses and irregular ones. For regular memory-access programs, we develop algorithms to precisely mark variables for invalidation in a core’s private cache — because they have become stale due to other cores’ writes. We also develop algorithms to accurately determine data that have to be written back from the private caches to shared caches because other cores need to access the data in future epochs.

For iterative irregular applications we present inspector-based schemes that exactly demarcate data for coherence. Other irregular parallel applications are handled via conservative methods that write back and invalidate data across synchronization points. These schemes do not preserve cache locality across synchronization points, but still enable data reuse in the cache by keeping read-only data if possible.

Compared to prior work on compiler-directed cache coherence [27, 25, 31], the compiler support developed in this chapter is more general. It is applicable to a larger class of programs, and the compiler analysis is more precise, as it takes into account the task-to-processor assignments.

The contributions of this chapter are:

- Compiler algorithms to automatically instrument parallel applications with cache management instructions that write back and invalidate cached data.

- For affine computations, algorithms to precisely identify data for cache coherence using the Polyhedral-model [41];
• Efficient techniques using the inspector-executor paradigm to ensure coherence for recurrent iterative irregular computations.

• An experimental demonstration using several programs that the compiler-based techniques we develop are competitive with hardware coherence schemes in terms of performance and energy consumption, at a lower hardware cost.

4.2 Overview

A computer system with software managed cache coherence does not implement cache coherence protocols in hardware. It is necessary to explicitly insert coherence instructions in a parallel program to ensure correct execution. The software orchestrates cache coherence using the following coherence primitives.

• **Writeback**: The address of a variable is specified in the instruction and if the addressed location exists in the private cache and has been modified, then it is written to a shared later level cache or main memory.

• **Invalidate**: The instruction causes any cached copy of the variable in the private cache to be discarded (self-invalidation) so that the next read to the variable fetches data from shared later level cache.

The above coherence operations provide a mechanism for two processors to communicate with each other: if processor A has to send an updated value of a shared variable X to processor B, then processor A issues a *writeback* instruction on X, and processor B later invalidates X so that a subsequent read to X fetches the latest value from the shared cache.
Fig. 4.1 shows the API for the invalidate and writeback instructions. These API functions use arguments at the granularity of word, double-word, or quad-word. The invalidate_range and writeback_range functions have a start address and number of bytes as parameters.

In this chapter, we address the question of how to automatically generate cache coherence instructions for execution on software managed caches. The variables that potentially hold stale data - invalidate set - have to be identified in order that copies of those variables in the private cache are discarded; the data that are produced at a processor, but might be consumed at other processors - writeback set - need to be characterized so that those data are written to the shared cache and are available to other processors for future reads.

In this section, we use an example to provide an overview of the analyses performed and optimizations applied by the algorithms developed in the chapter. Figure 4.2 shows a 1d-jacobi stencil code. In an iteration of the stencil computation, elements of array B are updated using three neighboring elements of array A in parallel (line 4). Then, all values of array B are copied to array A in parallel (line 8). In the next iteration, the new values of array A are used to update elements of array B. This process is repeated tsteps times (loop at line 1).

```c
invalidation_word(void *addr);
invalidation_dword(void *addr);
invalidation_qword(void *addr);
invalidation_range(void *addr, int num_bytes);

writeback_word(void *addr);
writeback_dword(void *addr);
writeback_qword(void *addr);
writeback_range(void *addr, int num_bytes);
```

Figure 4.1: Coherence API list
```c
for (t = 0; t <= tsteps - 1; t++) {
    #pragma omp parallel for
    for (i = 2; i <= n-2; i++) {
    }
    #pragma omp parallel for
    for (i = 2; i <= n-2; i++) {
        S2: A[i] = B[i];
    }
}
```

Figure 4.3: 1-d Jacobi stencil for SCC (unoptimized)

The parallelism in the computation is realized using OpenMP work-sharing constructs: the loops with iterators i are declared parallel (line 3 and 7), and different iterations of those loops may be assigned to different processors to execute them in parallel. The value written to B[i] by first statement (S1) is read by the second statement (S2). Since a different processor may execute the statement producing B[i] than those that consume B[i], B[i] has to be written-back to shared cache after S1 and is invalidated before S2.
for (t = 0; t <= tsteps - 1; t++) {
    #pragma omp parallel private(myid, i1, i2) {
        myid = omp_get_thread_num();
        for (i1 = myid; i1 < floor((n - 2) / 16); i1 += 8) {
            if (t >= 1) {
                invalidate_dword(&A[16*i1 - 1]);
                invalidate_dword(&A[16*i1 + 16]);
            }

            for (i2 = max(i1 * 16, 2); i2 <= min(i1 * 16 + 15, n - 2); i2 += 1) {
            }

            if (t == tsteps - 1) {
                writeback_range(&B[i1 * 16], sizeof(double)*16);
            }
        }
    }
}

#pragma omp parallel private(myid, i1, i2) {
    myid = omp_get_thread_num();
    for (i1 = myid; i1 < floor((n - 2) / 16); i1 += 8) {
        for (i2 = max(i1 * 16, 2); i2 <= min(i1 * 16 + 15, n - 2); i2 += 1) {
            S2: A[i2] = B[i2];
        }

        if (t <= tsteps - 2) {
            writeback_dword(&A[16*i1]);
            writeback_dword(&A[16*i1 + 15]);
        } else if (t == tsteps - 1) {
            writeback_range(&A[i1 * 16], sizeof(double)*16);
        }
    }
}

Figure 4.4: 1-d Jacobi stencil for execution on an 8-processor SCC system

Similarly, array element A[i] is written at S2 and has uses in three iterations of the first loop at S1. The reference A[i] must therefore be written-back after S2, and invalidated before S1. The code obtained thus for Software Cache Coherence (SCC) is shown in Figure 4.3.

However, there are opportunities to improve performance of the shown code on a software cache coherence system:
1. If iterations of the parallel loops in Figure 4.2 are mapped to processors such that an iteration with a certain value of \( i \) is assigned to the same processor in the first and second loops (line 3 and line 7), then \( B[i] \) is produced and consumed at the same processor. Therefore, writing back of \( B[i] \) at S1, and invalidating of it at S2 can be avoided, resulting in lower overhead and better cache locality (read misses to array \( B \) in the second loop will be avoided).

Thus, analysis to compute invalidation and writeback sets that considers the iteration-to-processor mapping can avoid many potentially conservative coherence operations.

2. Further, if iterations are mapped to processors in a block-cyclic manner, fewer invalidations and write-backs will be required. Consider for example that the iterations of the parallel loops are scheduled to processors in a block-cyclic manner with a chunk-size of 16. In that scenario, a processor writes to a consecutive set of words at S2 (line 8 in Figure 4.2) — from \( A[k+1] \) to \( A[k+16] \) (for some ‘k’) and the same processor in the next iteration reads elements \( A[k] \) to \( A[k+17] \) at S1 (due to line 4). Therefore, only \( A[k] \), and \( A[k+17] \) have to be invalidated. Equally, other processors would only reference \( A[k+1] \) and \( A[k+16] \), and hence, only those two array cells have to be written-back.

With the above considerations, one can proceed as follows.

1) By explicitly mapping iterations of the two parallel loops to processors through the use of \( myid \) and by pinning threads to cores, we can reduce the number of coherence operations;
2) The parallel iterations are distributed among processors in a block-cyclic manner.
The resulting code is shown in Figure 4.4. The very last write to a variable by any processor is also written-back so that results of the computation are available at the shared cache.

The algorithms that we develop in the chapter automatically produce the code in Figure 4.4 by performing a) an exact data dependence analysis (Section 4.4.1), b) iteration-to-processor mapping aware code generation (Section 4.4.2).

4.3 Background

4.3.1 Execution Model

Release Consistency: The execution of parallel programs consists of epochs (intervals between global synchronization points). In an epoch, data that were written potentially by other cores in previous epochs and that a core may need to read in the epoch are invalidated. Before the end of the epoch, all the data that a core has written in the epoch and that may be needed by other processors in future epochs are written-back to the shared level cache.

Before an epoch completes, all prior memory operations, including ordinary load/store instructions and coherence instructions, are completed. Then the next epoch can start, and the following memory operations can be initiated. Further, ordering constraints between memory instructions are respected: The order of a store to address $i$ and the following writeback for address $i$ should be preserved in the instruction pipeline of the processor and caches. Similarly, the order of invalidation to address $j$ and a following load from address $j$ should be preserved in the pipeline and caches to guarantee fetching of the value from the shared cache.

Coherence Operations at Cache line granularity: Coherence operations are carried out at the granularity of cache lines — all the lines that overlap with specified addresses are
invalidated or written-back. If the specified data are not present in cache, then coherence
instructions have no effect.

In addition, writeback instructions write back only dirty words of the line. In doing so,
writeback instructions avoid the incorrectness issue that may arise from false sharing: if
two processors are writing to variables that get mapped to the same cache line, and whole
cache lines (and not just the dirty words) are written-back, then one processor’s dirty words
may be overwritten with another processor’s clean words. Therefore, per-word dirty bits
are used to keep track of words of a cache line that are modified.

4.3.2 Notation

The code shown in Fig. 4.5 is used as a working example to illustrate the notation and
the compiler algorithm in the next section.

Sets: A set $s$ is defined as:

$$ s = \{ [x_1, \ldots, x_m] : c_1 \land \cdots \land c_n \} $$

where each $x_i$ is a tuple variable and each $c_j$ is a constraint.

The iteration spaces of statements can be represented as sets. For example, the iteration
space of statement $S_1$ in the code shown in Fig. 4.5 can be specified as the set $I^{S_1}$:

```c
for (t1 = 0; t1 <= steps - 1; t1++) {
#pragma omp parallel for private(t3)
    for (t2 = 0; t2 < n - 1; t2++) {
        for (t3 = 1; t3 < n - 1; t3++) {
            S1: B[t2][t3] = B[t2][t3+1] + 1;
        }
    }
}
```

Figure 4.5: A loop nest
\[ I^{S_1} = \{ S_1[t_1, t_2, t_3] : (0 \leq t_1 \leq t_{steps} - 1) \land (0 \leq t_2 \leq n - 1) \land (1 \leq t_3 \leq n - 1) \} \]

**Relations:** A relation \( r \) is defined as:

\[
r = \{ [x_1, \ldots, x_m] \mapsto [y_1, \ldots, y_n] : c_1 \land \cdots \land c_p \}
\]

where each \( x_i \) is an input tuple variable, each \( y_j \) is an output tuple variable and each \( c_k \) is a constraint.

Array accesses appearing in the code may be modeled as relations from iteration spaces to access functions of the array references. The two accesses to array ‘B’ in Fig. 4.5, \( B[t_2][t_3] \) and \( B[t_2][t_3+1] \), are represented as the following relations:

\[
r^{S_1}_{write} = \{ S_1[t_1, t_2, t_3] \mapsto B[t_2', t_3'] : (t_2' = t_2) \land (t_3' = t_3) \}
\]

\[
r^{S_1}_{read} = \{ S_1[t_1, t_2, t_3] \mapsto B[t_2', t_3'] : (t_2' = t_2) \land (t_3' = t_3 + 1) \}
\]

**The Apply Operation:** The apply operation on a relation \( r \) and a set \( s \) produces a set \( s' \) denoted by, \( s' = r(s) \) and is mathematically defined as:

\[
(\vec{x} \in s') \iff (\exists \vec{y} \text{ s.t. } \vec{y} \in s \land (\vec{y} \mapsto \vec{x}) \in r)
\]

The set of array elements accessed by an array reference in a loop (data-footprint) may be derived by applying access function relations on the iteration space sets. For the array accesses in the example code shown in Fig. 4.5, data-footprints of the two accesses are: \( r^{S_1}_{write}(I^{S_1}), r^{S_1}_{read}(I^{S_1}) \).

**The Inverse Operation:** The inverse operation \( r = r_k^{-1} \) operates on a relation \( r_k \) to produce a new relation \( r \) such that \( r \) has the same constraints as \( r_k \) but with the input and output tuple variables swapped. \( (\vec{x} \mapsto \vec{y} \in r) \iff (\vec{y} \mapsto \vec{x} \in r_k) \).
4.3.3 Polyhedral Dependences

In the Polyhedral model [125, 41], for affine computations, dependence analysis [42] can precisely compute flow (Read After Write - RAW) and output (Write After Write - WAW) dependences between dynamic instances of statements. The dependences are expressed as maps from source iterations to target iterations involved in the dependence.

The flow dependence determined by polyhedral dependence analysis (for example, using ISL [119]) for the code in Fig. 4.5 is:

\[ D_{\text{flow}} = \{ S1[t_1,t_2,t_3] \mapsto S1[t_1+1,t_2,t_3-1] : (0 \leq t_1 \leq tsteps - 2) \land (0 \leq t_2 \leq n - 1) \land (2 \leq t_3 \leq n - 1) \} \]

The relation characterizes the flow dependence that exists between the write reference \( B[t_2][t_3] \) and the read reference \( B[t_2][t_3+1] \).

An analysis tool like ISL can also be used to emit information regarding live-in data: data that are read in the loop but are not produced by any statement instances in the scope of analysis. A list containing maps from an iteration point that reads live-in data to the live-in array elements that it reads is computed. For the running example, the live-in maps are:

\[ D_{\text{live-in}} = \]

\[ \{ S1[0,t_2,t_3] \mapsto B[t_2,t_3 + 1] : 0 \leq t_2 \leq n - 1 \land 1 \leq t_3 \leq n - 2; \]

\[ S1[t_1,t_2,n - 1] \mapsto B[t_2,n] : 0 \leq t_1 \leq tsteps - 1 \land 0 \leq t_2 \leq n - 1 \} \]

The two maps capture live-in data read for read reference \( B[t_2][t_3+1] \): a) at the first iteration of the outermost \( t_1 \) loop, all read elements are live-in b) for later iterations, only
the \( n^{th} \) element in each row of matrix B is live-in, since it is never written to by the write reference \( B[t2][t3] \).

### 4.4 Compiler Optimization for Regular Codes

The iteration space of an epoch in a parallel loop is modeled by considering iterator values of the parallel loop and its surrounding loops as parameters. In the parallel loop in Fig. 4.5, the \( t2 \) loop is parallel and an iteration of \( t2 \) constitutes a parallel task executed in an epoch. Its iteration space is modeled by considering values of iterators \( t1 \) and \( t2 \) as parameters - \( t_p \) and \( t_q \) respectively:

\[
I_{current}^S = \{ S[t1,t2,t3] : (t1 = t_p) \land (t2 = t_q) \land (1 \leq t3 \leq n - 1) \}.
\]

#### 4.4.1 Computation of Invalidate and Writeback Sets

**Invalidate Set:** The invalidate set for a parallel task consists of data that are consumed in the parallel task but produced elsewhere. Thus, the invalidate set is the data consumed by those iterations of the parallel task that are targets of some flow dependence (RAW) whose source iterations are *not* in the same parallel task. The live-in data (first reads to any variable in the loop) are also invalidated and hence forms a part of the invalidate set.

Algorithm 3 shows how the invalidate set for a parallel task is computed. It is computed by forming the union of invalidate data sets corresponding to all statements within the parallel loop by iterating over each statement. For each statement \( S_i \), first the source iterations of the dependence - \( I_{source} \) - whose target iterations are in the current slice for that statement - \( I_{current}^S \) - are determined by applying the inverse relation of the flow dependence. From this set, any of the source iterations that lie in the current slice - \( \bigcup_{S_j \in \text{stmts}} I_{current}^S \) - are removed from \( I_{source} \) because the source and target iterations are run on the same processor and no coherence instruction is needed. The array elements written by iterations of \( I_{source} \)
are placed in the set of data elements for which invalidation coherence instructions must be issued to guarantee coherence. To this set is added the live-in list corresponding to data elements that come in live from outside the analyzed region.

Algorithm 3 Compute Invalidate Set

**Input:** Flow Dependences : $D_{\text{flow}}$, Live-in read maps : $D_{\text{live,in}}$, Current Iteration Slices: $I_{\text{current}}$, Write maps: $r_{\text{write}}$

**Output:** Statement and Invalidate set pairs: $D_{\text{invalidate}}^{S_i}$

1: for all statements - Si do
2: $D_{\text{invalidate}}^{S_i} \leftarrow \phi$
3: $I_{\text{source}} \leftarrow D_{\text{flow}}^{-1}(I^{S_i}_{\text{current}}) \backslash (\bigcup_{S_j \in \text{stmts}} I^{S_j}_{\text{current}})$
4: $D_{\text{inflow}} \leftarrow \bigcup_{S_j \in \text{stmts}} r_{\text{write}}^{S_j}(I_{\text{source}})$
5: $D_{\text{live,in,data}} \leftarrow D_{\text{live,in}}(I^{S_i}_{\text{current}})$
6: $D_{\text{invalidate}}^{S_i} \leftarrow (D_{\text{inflow}} \cup D_{\text{live,in,data}})$
7: end for

**Example:** The application of the algorithm to the running example results in the following invalidate set: $D_{\text{invalidate}}^{S_1} = \{[t, i_1] : 2 \leq i_1 \leq n\}$. The array elements read in the parallel task are marked for invalidation.

**Writeback Set:** The writeback set consists of all data that are produced by a parallel task and consumed outside of the parallel task. Thus we need to identify iterations of the parallel task that are sources of a flow dependence whose targets lie elsewhere. Once we have identified such source iterations, we then determine the array elements written by them. The last write to a variable also belongs to the writeback set and last writes are found by using output dependence (WAW) information: iterations which are not sources of any output dependence must be last writers to the array elements that they write to.

Algorithm 4 shows how we compute the writeback set for a parallel task that possibly has multiple statements in it. To find the writeback set corresponding to a statement $S_i$, 

88
first all target iterations \((I_{\text{target}})\) of all dependences are identified whose source iterations lie in \(I_{\text{current}}^S\). Those target iterations that are within the same parallel task - \(\bigcup_{S_j \in \text{stmts}} I_{\text{current}}^S\) are removed from \(I_{\text{target}}\) (line 3). Then the inverse dataflow relation is applied to this set and the intersection to the current iteration slice is computed (line 4) to identify the source iterations \((I_{\text{producer}})\) in the slice that write values needed outside this slice. These values must be part of the writeback set.

Further, if a write by an iteration is the last write to a certain variable, it must also be written back since it represents a live-out value from the loop. The iterations that are not sources of any output dependencies produce live-out values. Such iterations are determined by forming the set difference between \(I_{\text{current}}^S\) and domain of output dependences - \(\text{dom } D_{\text{output}}\).

**Algorithm 4 Compute Writeback Set**

**Input:** Flow Dependences : \(D_{\text{flow}}\), Output Dependences : \(D_{\text{output}}\), Current Iteration Slices: \(I_{\text{current}}\), Write maps: \(r_{\text{write}}\)

**Output:** Statement and Writeback set pairs: \(D_{\text{writeback}}^S\)

1. for all statements - \(S_i\) do
2. \(D_{\text{writeback}}^S_i \leftarrow \emptyset\)
3. \(I_{\text{target}} \leftarrow D_{\text{flow}}(I_{\text{current}}^S) \setminus (\bigcup_{S_j \in \text{stmts}} I_{\text{current}}^S)\)
4. \(I_{\text{producer}} \leftarrow D_{\text{flow}}^{-1}(I_{\text{target}}) \cap I_{\text{current}}^S\)
5. \(D_{\text{output}} \leftarrow r_{\text{write}}(I_{\text{producer}})\)
6. \(I_{\text{live\_out}} \leftarrow I_{\text{current}}^S \setminus \text{dom } D_{\text{output}}\)
7. \(D_{\text{live\_out\_data}} \leftarrow r_{\text{write}}(I_{\text{live\_out}})\)
8. \(D_{\text{writeback}}^S_i \leftarrow (D_{\text{output}} \cup D_{\text{live\_out\_data}})\)
9. end for

**Example:** The algorithm produces the following writeback set for the example in Fig. 4.5: \(D_{\text{writeback}}^S_i = \{ [t_q, i_1] : (t_p \leq t_{\text{steps}} - 2 \land 2 \leq i_1 \leq n - 1) \lor (t_p = t_{\text{steps}} - 1 \land 1 \leq i_1 \leq n - 1) \}\).
For 0 to tsteps-2 iterations of the outermost t1 loop, only elements B[t2][2:n-1] need to be written back as they will be read in the next iteration of t1 loop. Array cell B[t2][1] does not need to be written back because it is overwritten in a later t1 iteration and its value is not read. But the very last write to B[t2][1], i.e., when t1 = tsteps-1 has to be written back as it is a live-out value of the loop.

**Code Generation**  The invalidate and writeback sets are translated to corresponding cache coherence instructions by generating a loop to traverse elements of the sets using a polyhedral code generator — ISL [119]. The invalidations and writebacks are combined into coherence range functions whenever elements of a set are contiguous in memory: when the inner-most dimension of the array is the fastest varying dimension of the loop.

### 4.4.2 Optimization

**Analysis Cognizant of Iteration to Processor Mapping**  The techniques described until now do not assume any particular mapping of iterations to processors. However, if a mapping of processors to iterations is known, many invalidations and write-backs could possibly be avoided. For example, in the code shown in Fig. 4.5, the flow dependence (mentioned in §5.4.1) is: S1[t1,t2,t3] ↦ S1[t1+1,t2,t3−1]. If parallel iterations of the ‘t2’ loop are mapped to processors such that an iteration with a particular ‘t2’ value always gets mapped to the same processor, the source and target iterations of the flow dependence get executed on the same processor, making invalidations and write-backs due to the dependence unnecessary.

In order to incorporate this optimization, Algorithm 3 and 4 are modified to take iteration to processor mapping into account. Line 3 of Algorithm 3 is now changed to:

\[ I_{source} \leftarrow D_{flow}^{-1}(I_{current}^{Si}) \setminus (\bigcup_{Sj \in stmts} I_{current}^{Sj} \cup I_{same\_proc}) \]
and line 3 of Algorithm 4 is changed to:

$$I_{\text{target}} \leftarrow D_{\text{flow}}(I_{\text{current}}^{S_i}) \setminus (\bigcup_{S_j \in \text{stmts}} I_{\text{current}}^{X_j} \cup I_{\text{same}_\text{proc}}),$$

where $I_{\text{same}_\text{proc}}$ is the set of iterations that is executed on the same processor as the processor on which $I_{\text{current}}$ is executed.

For the working example, let us say that the OpenMP scheduling clauses specify that iterations are cyclically mapped onto processors and the number of processors used is 8. Then, we encode that information into the following iteration to processor map: $r_{t2p} = \{S1[t_1, t_2, t_3] \mapsto [t_2']: t_2' = t_2 \mod 8\}$. The parallel region code is all the iterations that are mapped to a parametric processor ‘myid’: $I_{\text{my}_\text{proc}} = r_{t2p}^{-1}(\text{myid})$. The iteration set $I_{\text{current}}^{S_1}$ is a subset of $I_{\text{my}_\text{proc}}$ with the values of the t1 and t2 loop iterators parameterized.

Using the modified algorithms, the cache coherence code generated for the working example is presented in Fig. 4.6. In the optimized code, only the live-in data is invalidated: elements $B[t2][2 \text{ to } n]$ at time-step $t_1 = 0$, only a single element – $B[t2][n]$ at later time-steps, since other elements are written to by the same processor ensuring that the updated values are present in the processor’s private cache. Only the live-out data is written back at the last time-step: $t_1 = t_{steps} - 1$.

**Iteration to Processor Mapping** A cyclic distribution of iterations to processors yields a better load balance especially for triangular iteration spaces; a block distribution of iterations to processors on the other hand, maps consecutive iterations to the same processor and hence reduces the amount of invalidations if flow dependences exist mainly between consecutive iterations. Further, if successive iterations write to consecutive locations in memory, a block distribution of iterations may enable a thread to collect write-backs together and perform coherence operations on a set of consecutive array elements.
for \( t1 = 0; t1 < \text{tsteps} - 1; t1++ \)
#pragma omp parallel private (myid, t2, t3) {
    myid = omp_get_thread_num();
    for (t2 = myid; t2 < n - 1; t2 += 8) {
        if (t1 == 0) {
            invalidate_range(&B[t2][2], sizeof(double) * (n - 2));
        }
        invalidate_dword(&B[t2][n]);
        for (t3 = 1; t3 < n - 1; t3++) {
            S1: B[t2][t3] = B[t2][t3 + 1] + 1;
        }
        if (t1 == tsteps - 1) {
            writeback_range(&B[t2][1], sizeof(double) * (n - 1));
        }
    }
}

Figure 4.6: Optimized loop nest for SCC

Therefore, for triangular iteration spaces, a block-cyclic distribution of iterations among processors is employed, for others a block distribution is performed.

4.5 Compiler Optimization for Irregular Codes

4.5.1 Basic Approach

The tasks that are executed in an epoch (interval between synchronization points) by construction do not have any dependences between them (otherwise, the dependences would induce serialization of tasks and hence, the tasks would have to be executed in different epochs). Therefore, all data accessed within an epoch can be safely cached and cache coherence is not violated.

```
invalidate_all();
writeback_all();
```

Figure 4.7: Coherence API for conservative handling
For irregular applications that have non-affine references and hence, are not amenable to the analysis presented in the previous section, software cache coherence is achieved conservatively: at the beginning of an epoch, the entire private cache is invalidated and at the end of the epoch, all data that are written in the epoch (dirty words) are written to the shared cache. The coherence API functions shown in Fig. 4.7 are inserted in the parallel program at epoch boundaries to conservatively manage software coherence.

The basic approach outlined above preserves intra-epoch cache data locality, but cannot exploit any temporal locality that exists across epoch boundaries.

## 4.5.2 Inspector-Executors

Many scientific applications use sparse and irregular computations and are often iterative in nature and furthermore, the data access pattern remains the same across iterations. (Examples include programs for solving partial differential equations, irregular stencils, the conjugate gradient method for solving systems of linear equations which uses sparse matrix-vector multiplications, atmospheric simulations that use semi-regular grids).

```c
while (converged == false)
{
    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        read A[B[i]]; /* data-dependent access */
    }

    #pragma omp parallel for
    for (i = 0; i < n; i++) {
        write A[C[i]]; /* data-dependent access */
    }
    /* Setting of converged variable not shown */
}
```

Figure 4.8: An iterative loop with irregular data references

93
For such codes, we propose the use of *inspectors* to gather information on irregular data accesses so that coherence operations are applied only where necessary. The inspectors that are inserted in the parallel codes are themselves parallel and are lock-free. The cost of inspectors is amortized by the ensuing selective invalidations of data and thus fewer unnecessary L1 cache misses over many iterations of the iterative computation.

Fig. 4.8 shows an iterative code that has data-dependent references to a one-dimensional array, viz., \( A[B[i]] \) and \( A[C[i]] \). We first illustrate the inspector approach for the simple example. The ideas are more generally applicable in the presence of multiple arrays and multi-dimensional arrays.
/* Inspector code begins */
#pragma omp parallel for
for (i=0;i<n;i++) {
    A_thread[i] = -1;
    A_conflict[i] = 0;
    writeback_word(&A_thread[i]);
    writeback_word(&A_conflict[i]);
}
// Phase 1: Record writer thread ids
#pragma omp parallel for
for (i=0;i<n;i++) {
    A_thread[C[i]] = myid;
    writeback_word(&A_thread[C[i]]);
}
// Phase 2: Mark conflicted if writer and reader threads are not the same
#pragma omp parallel for
for (i=0;i<n;i++) {
    invalidate_word(&A_thread[B[i]]);
    if (A_thread[B[i]] != -1 & A_thread[B[i]] != myid) {
        A_conflict[B[i]] = 1;
        writeback_word(&A_conflict[B[i]]);
    }
}
/* Inspector code ends */
#pragma omp parallel
{ invalidate_all(); }
while (converged == false)
{
    #pragma omp parallel for
    for (i=0;i<n;i++) {
        if (A_thread[B[i]] != -1 & A_thread[B[i]] != myid)
            invalidate_word(&A[B[i]]);
        read A[B[i]];
    }
    #pragma omp parallel for
    for (i=0;i<n;i++) {
        write A[C[i]];
        if (A_conflict[C[i]] == 1)
            writeback_word(&A[C[i]]);
    }
    /* Setting of converged variable not shown */
}
#pragma omp parallel
{ writeback_all(); }

Figure 4.9: An iterative loop with irregular data references for SCC system
The inspector-code determines if a) the write performed at a thread has readers at other threads: if that is the case, the variable has to be written-back to shared cache so that other threads will be able to obtain the updated value of the variable. b) the variable being read at a thread was written by another thread: if yes, the variable has to be invalidated at the private cache so that the fresh value is got from shared cache.

Fig. 4.9 presents the inspector-inserted parallel code corresponding to the iterative loop shown in Fig. 4.8 for execution on software managed caches. Two shadow arrays — A_thread and A_conflict for array A that has data-dependent accesses are initialized (lines 4, 5). In the first phase, A_thread records the ids of the threads that write to array cells (line 13). In the second phase, if an array cell is read by a thread different from the writer thread, the corresponding cell in A_conflict array is set to 1 (line 23). Since the computation loops are parallel, the inspection is also carried out in parallel. Consequently, accesses to arrays A_thread and A_conflict are guarded with coherence instructions. If there are multiple readers for an array cell then more than one thread may set the respective cell of A_conflict to 1 in phase two and multiple threads will write-back the same value, namely 1 to shared cache (in line 24). Since the same value is being written, any ordering of writes by different threads works.

Later in the computation loops, a thread invalidates a variable (line 37) before reading it if the variable has a writer (as opposed to read-only data) and that writer is a different thread. A thread after writing to a variable, writes it back (line 47) if the variable is marked conflicted.
4.5.3 Exclusion of Read-Only Data from Coherence

For irregular codes whose data access patterns potentially change with each iteration, we adopt a conservative approach that yet excludes read-only data from coherence enforcement and thus, is more accurate than a full invalidation and writeback approach outlined earlier.

We consider parallel regions — parallel loops along with surrounding looping structures and perform analysis of the parallel region as a stand-alone unit. The read-only data of the parallel region need not be invalidated/written-back. Only those variables that are both written and read in the parallel region are invalidated and written-back at epoch boundaries.

For this scheme to work however, the following conditions have to be met:

1. None of the processors should have cached stale values of read-only data of the parallel region. (This could happen for example when, a program has a parallel
region $\mathcal{P}$ followed by a sequential segment $\mathcal{Q}$ and later a parallel region $\mathcal{R}$. And, variable $x$ is read-only in $\mathcal{P}$ and $\mathcal{R}$, but is modified in $\mathcal{Q}$.

2. Since, in the parallel region coherence is enforced only on data that are both read and written, for written-but-not-read data coherence operations should be introduced following the parallel region to ensure that future accesses to them get updated values.

To meet condition 1), a prologue is introduced that writes back all dirty words from the master thread and then does a full invalidation of caches at all threads. Condition 2) is fulfilled by writing-back all dirty words from all threads and doing a full-invalidation by the master thread in an epilogue. The code shown in Fig. 4.10 uses the outlined approach.

**Algorithm 5** Generate Coherence Instructions using Parallel Region Analysis

**Input:** AST of Parallel region: $\mathcal{P}$

**Output:** AST of Parallel region for SCC: $\mathcal{P}_{SCC}$

1. $\textit{Prologue} \leftarrow$ API to write-back all dirty words from master thread; API to invalidate entire cache of all threads
2. $\textit{Read Set} \leftarrow$ Arrays and scalars that are read in $\mathcal{P}$
3. $\textit{Write Set} \leftarrow$ Arrays and scalars that are written in $\mathcal{P}$
4. $\textit{Coherence Set} \leftarrow \textit{Read Set} \cap \textit{Write Set}$
5. for all epoch code $e \in \mathcal{P}$
6. $\textit{Invalidate Set}_e \leftarrow \textit{Read Set}_e \cap \textit{Coherence Set}$
7. $\textit{Writeback Set}_e \leftarrow \textit{Write Set}_e \cap \textit{Coherence Set}$
8. Insert API for $\textit{Invalidate Set}_e$ and $\textit{Writeback Set}_e$
9. end for
10. $\textit{Epilogue} \leftarrow$ API to write-back all dirty words from all threads; API to invalidate entire cache of master thread
11. $\mathcal{P}_{SCC} \leftarrow$ Append $\{\textit{Prologue}, \mathcal{P}, \textit{Epilogue}\}$

Algorithm 5 presents the overall parallel-region analysis technique.
4.6 Experimental Evaluation

We evaluate the performance of compiler-generated coherence instructions for execution of parallel programs on software managed caches. The main goal of the compiler support developed in the chapter is to insert coherence instructions — invalidate and write-back functions only where necessary. The conservative invalidations (of non-stale data) result in read misses which lead to degraded performance relative to a hardware coherence scheme. Therefore, to assess efficacy of the compiler techniques, we compare read misses in L1 caches, and execution time on software and hardware managed caches. (The number of misses at the shared cache is unaffected and will be the same for software and hardware cache coherence.)

Conservative coherence operations in software scheme increase accesses to the shared cache and also, cause increased traffic on the system bus. The hardware cache coherence protocol uses control messages to maintain coherence, which a software scheme does not. Therefore, if the software coherence mechanism results in comparable cache misses as a hardware protocol then, the software coherence also reduces network traffic and cache energy. We therefore measure the number of words transferred on the system bus and cache energy by software and hardware coherence systems.

4.6.1 Benchmarks

The benchmark programs listed in Table 4.1 are used for the experiments. The benchmark codes are from Rodinia [23] and PolyBench [94] benchmark suites. The Rodinia suite provides parallel programs from various application domains. The backprop, hotspot, kmeans, pathfinder, srad applications are taken from Rodinia suite, and they contain affine as well as irregular data references. The PolyBench benchmark suite is a collection of
Table 4.1: Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gemm</td>
<td>Matrix-multiply: ( C = \alpha A \cdot B + \beta C )</td>
</tr>
<tr>
<td>gemver</td>
<td>Vector Multiplication and Matrix Addition</td>
</tr>
<tr>
<td>jacobi-1d</td>
<td>1-D Jacobi stencil computation</td>
</tr>
<tr>
<td>jacobi-2d</td>
<td>2-D Jacobi stencil computation</td>
</tr>
<tr>
<td>LU</td>
<td>LU decomposition</td>
</tr>
<tr>
<td>trisolv</td>
<td>Triangular solver</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate Gradient method</td>
</tr>
<tr>
<td>backprop</td>
<td>Pattern recognition using unstructured grid</td>
</tr>
<tr>
<td>hotspot</td>
<td>Thermal simulation using structured grid</td>
</tr>
<tr>
<td>kmeans</td>
<td>Clustering algorithm used in data-mining</td>
</tr>
<tr>
<td>pathfinder</td>
<td>Dynamic Programming for grid traversal</td>
</tr>
<tr>
<td>srad</td>
<td>Image Processing using structured grid</td>
</tr>
</tbody>
</table>

widely used linear algebra, and stencil codes. The benchmark programs — gemm, gemver, jacobi-1d, jacobi-2d, LU, trisolv are taken from PolyBench suite. The codes are parallelized using a polyhedral compiler – PoCC [93]. All array references in the PolyBench programs are affine.

4.6.2 Set-up

The snooping-bus MESI protocol hardware coherence (referred to as HCC in the following text), and software cache coherence (referred to as SCC) have been implemented in an architectural multi-processor simulator — SESC [88]. Details of the simulator setup are described in Table 4.2.

We compare performance and energy of the following four coherence schemes:

1. **HCC**: Parallel programs are executed using MESI hardware coherence.
Table 4.2: Simulator parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor chip</td>
<td>8-core multicore chip</td>
</tr>
<tr>
<td>Issue width; ROB size</td>
<td>4-issue; 176 entries</td>
</tr>
<tr>
<td>Private L1 cache</td>
<td>32KB Write-back, 4-way, 2 cycle hit latency</td>
</tr>
<tr>
<td>Shared L2 cache</td>
<td>1MB Write-back, 8-way, multi-banked, 11 cycle round-trip time</td>
</tr>
<tr>
<td>Cache line size</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Cache coherence protocol</td>
<td>Snooping-based MESI protocol</td>
</tr>
<tr>
<td>Main Memory</td>
<td>300 cycle round-trip time</td>
</tr>
</tbody>
</table>

2. **SCC-basic**: The coherence instructions are inserted without iteration-to-processor aware analysis for affine references and without the use of inspector-executor or read-only data exclusion scheme for irregular accesses. That is, coherence instructions are generated with methods described in Sections 4.4.1 and 4.5.1 only without further optimizations. The resulting codes are run on software managed caches.

3. **SCC-opt**: The coherence management is optimized using compiler optimizations presented, and the resulting programs are executed on software managed caches.

4. **HCC-opt**: To study if any optimizations applied to SCC codes (such as explicit mapping of iterations to processors) can also benefit the benchmarks for hardware coherence, SCC-opt programs are adapted to run on HCC systems: coherence operations and any inspectors inserted are removed from SCC-opt codes and these variants are run on the HCC system.

The performances of only parallel parts of benchmarks are measured — sequential initialization and finalization codes are excluded from measurements because the performance
of sequential code is expected to be the same on SCC and HCC systems. Threads are pinned to cores for both schemes.

4.6.3 Performance Results

![L1 Data Cache Read Misses](image)

Figure 4.11: L1 data cache read misses (lower, the better). The L1 read miss ratios for HCC are also shown.

Fig. 4.11 plots the read misses in L1 cache; Fig. 4.12 shows the execution time. The number of L1 read misses and execution cycles are normalized with respect to HCC statistics (the number of misses and execution cycles of HCC is considered 1). The L1 read miss ratios (fraction of L1 reads that are misses) for HCC are also indicated in the graph.

On average (geometric mean) across benchmarks, HCC-opt has the same number of cache misses as HCC; SCC-basic suffers 98% more misses and SCC-opt experiences only a 3% increase (avg. column in the graph). The geometric mean of normalized execution time for the three variants — HCC-opt, SCC-basic, and SCC-opt are, 0.97, 1.48, and 0.97
respectively. We observe that SCC-opt greatly improves performance over SCC-basic and brings down cache misses comparable to those of HCC. Further, performance of HCC-opt is very similar to that of HCC.

The gemm and trisolv benchmarks exhibit the so-called *communication free* parallelism: the outer loop in these codes is parallel. Therefore, there is no communication between processors induced by data dependences. All code variants of gemm and trisolv have virtually the same number of cache misses and execution cycles.

In applications that have irregular references, namely backprop, CG, hotspot, kmeans, pathfinder, srad, the parallel region boundaries are guarded with full-invalidation and full-writeback instructions (described in 4.5.3) The affine accesses in the parallel regions are optimized; irregular accesses are handled using *inspectors* or invalidation and write-back of entire arrays that are both written and read in the parallel region (read-only arrays and scalars are excluded).
For backprop and pathfinder, full invalidation of cache at parallel region boundaries results in some loss of data locality which results in increased L1 cache read misses.

The CG and srad benchmarks have iterative loops and irregular accesses whose indexing structures do not change across iterations. Therefore, for those two benchmarks, inspector codes are inserted for deriving coherence operations. The inspectors contribute to a certain number of L1 read misses. The reduced cache misses in SCC-opt of srad compared to its HCC counterpart is an artifact of the interaction that exists between cache coherence and cache replacement policy (LRU): false-sharing in HCC can cause soon-to-be-reused data to be evicted, which favors SCC. The migratory writes may sometimes cause invalidations of not-to-be-reused data and thus, making way for other to-be-reused data and this benefits HCC. Conversely, the gemver is an example of hardware cache coherence working to HCC advantage, where the number of misses for HCC is lower compared to SCC-opt.

The running time (depicted in Fig. 4.12) shows a strong correlation between L1 cache read misses and performance. In HCC, the snooping overhead plays a significant role in determining execution time: In our implementation, we assign 1 cycle to a read/write snooping request. In SCC, each coherence instruction incurs a two-cycle overhead. In addition to these overheads, there may be additional overheads depending upon the response to a snooping request in HCC (e.g., a read request may return an updated value from another processor) and the number of cache lines specified in the coherence instruction in SCC — each cache line incurs a 2-cycle delay. Because of removal of hardware cache coherence, we observe a 3% performance gain for SCC-opt over HCC on average.

**Discussion:** The performance results obtained for HCC and SCC schemes are sensitive to architectural choices made in the simulator implementation. And, we have opted for architectural choices that favor HCC even though on a real system they may be impractical.
or too costly. E.g., we have allotted 1-cycle delay for a snooping request and on a real system it might take multiple cycles. The implemented HCC protocol in the simulator concurrently sends a snoop request to other cores, and also a memory request to L2 cache. Alternately, the L1 cache can also be designed to send a memory request to L2 cache after a snoop miss, but this will increase the delay when there is a snoop miss.

4.6.4 Energy Results

Figure 4.13: Traffic on the system bus (lower, the better). Average number of words per cycle for HCC is also shown.

**Bus data transfers:** Fig. 4.13 shows the traffic (number of words transferred) on the system bus for different schemes. All values are normalized with respect to HCC. The average number of words transferred per cycle (obtained by dividing total number of words with number of execution cycles) for HCC is also shown. For hardware coherence scheme, the traffic on the bus includes snoopy-bus coherence related exchange of messages,
transfers between private L1 caches and shared L2 cache triggered by cache misses at L1 and replacement of cache lines at L1. For SCC, this includes data transfers between L1 caches and L2 cache prompted either by L1 misses and evictions, or invalidation and writeback coherence instructions. The HCC normalized data transfers on the bus for HCC-opt, SCC-basic, and SCC-opt are 0.99, 1.46, and 0.99 respectively, on average (geo-mean).

In backprop and srad, SCC-opt does a fewer write-backs to L2 cache compared to HCC; the L1 cache misses are lower for SCC-opt in the case of srad. Consequently, SCC-opt incurs a fewer data transfers in backprop and srad. Conservative writebacks in kmeans increases the traffic on the bus for SCC-opt compared to HCC.

**L1 and L2 cache energy:** The cache SRAM is a major consumer of energy in a processor. We compare cache energy consumption for HCC and SCC-opt schemes based on the number of accesses to tag SRAMs and data SRAMs. Using the SESC simulator, event counts for all relevant activities in the L1 and L2 caches are collected to account for all tag
and data accesses to SRAMS. CACTI [122] is used to obtain the energy per access for tag and data for each cache level. The L1 cache employs dual ported SRAM to service snoop requests quickly. For SCC also we used the same dual ported SRAM for a fair comparison (per-access cost is a function of, inter alia, number of ports). The L1 cache accesses tag and data together for local processor requests while for snooping requests it accesses data SRAM only after tag hit. The L2 cache is configured to be in sequential access mode — it starts to access data SRAM after tag matching. We did not consider main memory energy because main memory accesses would be the same for both HCC and SCC schemes.

Fig. 4.14 plots relative energy consumption in caches for hardware and software cache coherence approaches: energy expenditure by HCC is considered 1 and energy dissipation by SCC-opt is scaled with respect to HCC. The break-down of energy expended in L1 and L2 caches is indicated. On average (arithmetic mean) SCC-opt energy consumption in caches is 5% less than that of HCC.

Most of the savings in SCC-opt come from two sources: elimination of snooping requests in L1 cache, and reduction in the number of writeback words by partial line transfers (only dirty words are written back to shared L2 cache in a software managed cache as opposed to entire cache lines which are the granularity of communication for HCC). We also observe that energy spent in all L1 caches together is around 86%, while the rest — 14% is expended in L2 cache.

4.7 Related Work

Some prior studies [27, 25, 31] have developed compiler analysis techniques to generate cache coherence instructions for software managed caches. The work in this chapter
distinguishes itself from prior efforts both by being more general as well as more precise, as we elaborate below.

Cheong et al. [25] use data flow analysis to classify every reference to shared memory either as a memory-read or a cache-read. A read reference is marked as a memory-read if the compiler determines that the cache resident copy of the data might have become stale, otherwise the reference is marked as cache-read. A limitation of that work is that the data flow analysis is carried out at the granularity of arrays, which will result in invalidations for an entire array even if two processors are accessing distinct parts of it.

Choi et al. [27] propose to improve inter-task locality in software managed caches by using additional hardware support: the current epoch number is maintained at runtime using an epoch counter and each cache word is associated with a time-tag which records the epoch number in which the cache copy is created. Then they develop the so-called epoch flow graph technique to establish conditions under which it can be guaranteed that the cached copy of a variable is not stale. The analysis here too treats an entire array as a single variable.

Darnell and his colleagues [31] perform array subscript analysis to gather more accurate data dependence information and then aggregate cache coherence operations on a number of array elements to form vector operations. The method however can handle only simple array subscripts: only loop iterators are allowed as subscripts.

O’Boyle et al. [85] develop techniques to identify data for Distributed Invalidation (DI) for affine loops that are run on distributed shared memory architectures. The DI scheme uses a directory to maintain coherence, and where possible it seeks to eliminate invalidation messages from the directory to remote copies and associated acknowledgments. Their analysis to minimize invalidation messages has similarities to our analysis for minimize
invalidations. But the coherence equations in DI place some restrictions on the kinds of affine loops that can be analyzed: for example, conditional execution within the loop is disallowed, and increments of loop iterators must be unity. The approach presented in this chapter efficiently handles arbitrary affine loops including those whose iterator values are lexicographically decreasing, that have a non-unit trip-count, or have a modulus operator etc., and conditionals are permitted. The DI work does not involve writebacks, which however are a part of our software cache coherence model, and we develop techniques to optimize writebacks as well. We also optimize irregular codes using an inspector-executor approach, while such codes are not optimized in the DI scheme.

Inspector-executor approaches have been used in the context of parallelization [32, 99, 56, 133] run-time reorderings [35, 52], but to our knowledge have not previously been developed for optimizing for cache coherence.

Kaxiras et al. [65] seek to improve scalability of directory coherence by creating tear-off copies: the read-only data are cached in the private cache of a core, but are not registered in the directory. And, at the first synchronization event, non-registered copies are self-invalidated without generating invalidation traffic.

Kontothanassis et al. [69] present a software cache coherence protocol with page granularity in large scale machines. Our approach differs from their work in that fine-grained sharing in on-chip multi-core processors is accurately handled with compile-time analyses. Ashby et al. [6] propose a software cache coherence scheme that uses a bloom filter to avoid unnecessary invalidations, but their work does not develop any compiler support. DeNovo [26] simplifies complicated hardware cache coherence protocols by enforcing a disciplined parallel programming model. The compiler support proposed in this work can complement the DeNovo project in automatically identifying self-invalidation regions.
4.8 Conclusion

The complexity of developing efficient hardware coherence protocols for emerging manycore heterogeneous systems makes software controlled coherence schemes attractive. However, a significant challenge for software controlled cache coherence is that of generation of efficient coherence instructions.

The automatic coherence management and optimization approaches developed in the chapter advance compiler technology towards making software cache coherence a viable solution on shared-memory multiprocessor systems. Simulation results demonstrate the effectiveness of the compiler algorithms in achieving performance and cache-energy comparable to that of a hardware cache coherence scheme.
Chapter 5: Compiler-Assisted Detection of Transient Memory Errors

5.1 Introduction

Trends in technology scaling have increased the likelihood of transient faults in various hardware components due to particle strikes [90, 15, 10, 97] and environmental factors [80, 134]. The simultaneous drive to reduce power consumption has led to the use of lower voltage levels and smaller noise margins, which further increase a system’s susceptibility to transient faults [38]. Such transient faults in the memory subsystem can result in bit flips that are not detected and potentially lead to silent data corruption.

Hardware approaches to detecting and correcting bit flips in the memory system employ error correcting codes on various components of the data path. These codes are checked on every data access and updated on every modification. Comprehensive error detection requires every component of the data path – reorder buffers, caches, memory lines, buses, etc. – to support sufficient redundancy or parity to detect the errors anticipated during execution. Such a design requires pre-allocation of hardware resources, incurring dynamic power and latency costs to support the worst case fault scenario to be tolerated. Commodity computing platforms with less-protected memory subsystems might nevertheless have to contend with faults for certain critical computation phases. Even custom computing platforms more cognizant of soft errors might not provide the same level of protection in all components of
the memory subsystem. Examples include L1 caches in BlueGene/L [72], GPU memory on pre-Fermi Nvidia GPUs [82] and pre-Tahiti AMD GPUs with parities. Multi-bit errors can often evade hardware detection mechanisms and potentially lead to silent data corruption. In addition to multiple bit flips in stored data, an error in the addressing logic in the memory subsystem, including in address generation, might result in an incorrect address and be perceived as a multi-bit error. More importantly, the fault scenarios encountered in practice might not always match the fault models and projections assumed in the design phase.

In this chapter, we consider software-level approaches to comprehensive detection of multi-bit errors in the memory subsystem. Such approaches can complement hardware schemes to further improve system resilience. Redundant execution of memory operations which duplicates all variables of interest and operations on them can be used to detect such errors in the memory subsystem. However, this basic approach significantly increases memory space and bandwidth requirements. We study the feasibility of detecting errors by employing error detection codes for the definition and every use of variables. This approach has the potential to comprehensively detect errors due to faults in any architectural state in the memory subsystem.

We present a compiler-assisted approach to augmenting the definitions and uses in a given program with checksums to detect memory errors. We present optimizations that minimize the overhead for common classes of computations: affine loops and iterative computations that could be irregular. Experimental evaluation demonstrates that the performance costs are low enough for the approach to be practical and that we achieve excellent
fault coverage for the checksum operator considered. We also demonstrate that the overheads can be reduced further with hardware support in the processor (without affecting or altering the memory subsystem) to compute the checksums.

The primary contributions of this chapter are:

- an algorithm to detect memory errors by augmenting definitions and uses of values with checksum computation operations;
- compile-time optimizations to make this approach more efficient in two common classes of computations;
- a discussion and evaluation of the checksum operator in terms of performance and fault coverage;
- novel proposals to increase fault coverage via use of multiple checksums; and
- experimental demonstration of the low overheads and feasibility of the approach.

5.2 Error Detection using Checksums

Typical approaches to protecting data elements in hardware and software group data elements and employ a checksum for each group. These checksums are typically checked on every access. Algorithm-based fault tolerance approaches employ checksums for matrices which are maintained as part of the algorithm execution. All these schemes attempt to associate checksums with data elements. This requires additional steps to maintain the checksum for every operation on the data element. In architectural checksumming instantiations, this requires every storage element to be sufficiently protected by checksums.
5.2.1 Software Approach

In this chapter, we present a software approach to checksums on the definition and use of variables. Consider the code listing in Figure 5.1(a). The first statement adds two constants, 10 and 20, and stores the result in variable temp. The second and third statements use the value stored in temp to perform their computations. Between the definition of temp and its subsequent uses, temp is susceptible to memory errors. We seek to verify that the value stored in temp at the time of its definition is indeed what is used in all subsequent uses. The checksum approach is illustrated in Figure 5.1(b). The definition of temp contributes to a definition checksum. Each use of temp contributes to a use checksum. In addition, the number of uses of the variable temp is tracked. At program termination, or at any post-dominator of all definitions and uses tracked, we verify that the definition checksum scaled by the tracked number of uses equals the use checksum.

Such a def-use based checksum provides comprehensive coverage and can detect faults irrespective of the architectural characteristics of the memory subsystem. However, use of such a checksum scheme in practice requires several challenges to be overcome. Scaling the definition checksums with the use counts for each value potentially requires checksums for each value to be individually stored. This dramatically increases memory space and bandwidth overheads. Maintaining the use count information itself can introduce significant overheads. In addition, the checksum computation introduces an arithmetic operation for every definition and use, i.e., for every load and store instruction.

In the rest of the chapter we address the following challenges:

- How can the number of uses for a given definition be efficiently determined?

- How can the checksums be encoded to minimize the memory costs?
• How can the checksum operators be chosen to minimize the computation costs and maximize fault coverage?

• What is the overhead associated with these schemes?

As system architectures evolve, greater amounts of compute resources are available as compared to memory resources. We therefore also consider the possibility of hardware support in the processor to assist in the checksum computation.

5.2.2 Fault Model

We consider undetected and uncorrected errors in the memory subsystem. This includes (a) undetected multi-bit errors in main memory, caches, write queues, etc.; and (b) errors in address generation that result in incorrect data location being operated upon. We focus on incorrect memory operations that go undetected and could potentially lead to silent data corruption. Note that reading data from incorrect locations can cause several bits to differ from the expected value.

The control flow variables such as loop indices are assumed to be protected through other means (duplication, invariant assertions, special hardware, etc.). We assume that the processor’s subsystems other than memory are resilient to faults. This includes registers, ALUs, pipeline latches, and other logic. A consumed value is assumed resilient once it enters the processor, and conversely a produced value is assumed correct until it is written out by a store instruction. We therefore focus on detecting errors in a variable between the time it is written and later read.
temp = 10 + 20  
contrib_def_chksum(temp)

sum1 = temp + 30  
contrib_use_chksum(temp)
inc_use_count(&temp)

sum2 = temp + 40  
contrib_use_chksum(temp)
inc_use_count(&temp)

/* verify def_checksum scaled by the use counts matches the use_checksum */

(a) Original code

(b) Checksum augmented code

Figure 5.1: Illustration of insertion of error detection codes

5.3 Overview of the Solution

We use integer modulo addition as the checksum operator in this work. Two global checksums are used: i) \texttt{def} \_\texttt{checksum}, to accumulate (using integer modulo addition) all assigned values to data elements, and, ii) \texttt{use} \_\texttt{checksum}, to accumulate (again via integer modulo addition) all uses of all data elements. For each instruction, all input operands are added to \texttt{use} \_\texttt{checksum}; and the output is added to \texttt{def} \_\texttt{checksum}, if the output is going to be used later on.

For the code shown in Figure 5.1(a), variable \texttt{temp} defined at the first statement can be statically analyzed by the compiler (Section 5.4) that it is used in two subsequent statements. Therefore it is multiplied by two and added to \texttt{def} \_\texttt{checksum}, so that its contribution to \texttt{def} \_\texttt{checksum} will exactly match the total contribution to \texttt{use} \_\texttt{checksum} from the two uses of \texttt{temp} later on, if we have error-free execution. Variables \texttt{sum1}, \texttt{sum2} are defined
temp = 10 + 20  

def_checksum += 2*temp

sum1 = temp + 30  

use_checksum += temp

sum2 = temp + 40  

use_checksum += temp

assert(def_checksum==use_checksum)

(b) EDC added code

(a) Original code

Figure 5.2: Resilient Code

but not used in the code, and hence their use-count is zero. Therefore, they are not added to \texttt{def checksum}. At the end of the code region, an assertion statement is inserted, asserting equality of \texttt{def checksum} and \texttt{use checksum} to establish absence of data corruption faults. If the two checksums do not match, that indicates that program execution has been affected by memory errors. The resulting resilient code is presented in Figure 5.2(b).

We note that the above approach addresses the challenges noted in Section 5.2:

- An associative and commutative checksum operator such as integer modulo addition is used;

- The produced value is multiplied by the number of times it will be used — \texttt{use count}, and added to the definition checksum.

Because of the above scheme, \textit{checksums on a per-variable basis are not needed, rather, only two global checksums can be used to track definition and use of all variables.}
for $j = 0$ to $n-1$

$S1$: $A[j][j] = \sqrt{A[j][j]}$

for $i = j+1$ to $n-1$


Figure 5.3: Example affine code snippet

In scenarios where use_count of a variable cannot be determined at compile-time due to data-dependent nature of control flow or data accesses, counter variables are used to keep track of the number of uses and checksums are adjusted at the end of the live-range of the variable (described in detail in Section 5.5).

5.4 Compile-time Determination of Use Counts for Affine References

In this section, we describe the algorithm for determining the use counts at compile-time for affine references, beginning with the mathematical notation used.

5.4.1 Notation

We use the same notation as used by Verdoolaege [119] for the definition of sets, relations, apply, and inverse operations.

Sets. A set $s$ is defined as:

$$s = \{ [x_1, \ldots, x_m] : c_1 \land \cdots \land c_n \}$$

where each $x_i$ is a tuple variable and each $c_j$ is a constraint.

The iteration spaces of statements (the set of iteration points of the loop) can be represented as sets. For example, the iteration space of statement $S1$ in the code shown in
Figure 5.3 can be specified as the set $I^{S_1}$:

\[ I^{S_1} = \{ S_1[j] : (0 \leq j \leq n - 1) \} \]

And, the iteration space of statement $S_2$ is represented as the set

\[ I^{S_2} = \{ S_2[j,i] : (0 \leq j \leq n - 1) \land (j + 1 \leq i \leq n - 1) \} . \]

**Relations.** A relation $r$ is defined as:

\[ r = \{ [x_1,\ldots,x_m] \mapsto [y_1,\ldots,y_n] : c_1 \land \cdots \land c_p \} \]

where each $x_i$ is an input tuple variable, each $y_j$ is an output tuple variable and each $c_k$ is a constraint.

The read and write references of loops are specified as relations from iteration points to array indexes. For example, the write reference $A[j][j]$ in statement $S_1$ of the code shown in Figure 5.3 is characterized as:

\[ r^{S_1}_{\text{write}} = \{ S_1[j] \mapsto A[j,j] \} . \]

Data dependences appearing in the program are also expressed as relations between statements. For example, the flow (Read After Write or RAW) dependence between write reference $A[j][j]$ of statement $S_1$ and read reference $A[j][j]$ of statement $S_2$ of the example code is represented by the following relation:

\[ d_{\text{flow}} = \{ S_1[j] \mapsto S_2[j,i] : (0 \leq j \leq n - 1) \land (j + 1 \leq i \leq n - 1) \} . \]

**The Apply Operation.** The apply operation on a relation $r$ and a set $s$ produces a set $s'$ denoted by, $s' = r(s)$ and is mathematically defined as:

\[ (\vec{x} \in s') \iff (\exists \vec{y} \text{ s.t. } \vec{y} \in s \land (\vec{y} \mapsto \vec{x}) \in r) \]
For a given source iteration of a dependence, its target iterations can be found by applying the dependence relation on the given source iteration. For example, consider the source iteration \( si = \{ S1[10] \} \), i.e., the dynamic instance of \( S1 \) when the value of loop iterator \( j \) is 10. Its target iterations due to the above flow dependence – \( d_{flow} \) – can be determined as follows:

\[
d_{flow}(si) = \{ S2[10,i] : 11 \leq i \leq n - 1 \}.
\]

**Schedules.** The order of execution of statements in a given program is encoded using 2d+1 schedules [45], where ‘d’ is the maximum number of loops surrounding any statement. A schedule maps iterators of a statement to a combination of iterators and scalar values that specify a global ordering of statements within the program. The abstract syntax tree (AST) of the given program is used to deduce schedules as follows. Each edge of the AST is assigned a number: edges originating from a node are numbered from left to right, starting from 0). The schedule of a statement is the vector of numbers and iterators from the root of the AST to the statement node. If the dimensionality of a schedule is less than 2d+1, it is extended to dimensionality 2d+1 by filling the trailing components of the schedule vector with zeroes.

The AST for the example code is shown in Figure 5.4. and edges of the AST are numbered according to the rule described above. The schedules for statements \( S1 \) and \( S2 \) for the example in Figure 5.3 are shown below.

\[
\begin{align*}
S1[j] & \mapsto [0, j, 0, 0, 0] \\
S2[j, i] & \mapsto [0, j, 1, i, 0]
\end{align*}
\]

**Polyhedral Dependences.** In the polyhedral model [41] for affine computations, dependence analysis [42] can precisely compute flow (Read After Write or RAW) dependences
between dynamic instances of statements. The dependences are expressed as relations from a source iteration to its target iterations involved in the dependence.

Polyhedral dependence analysis (for example, using ISL [62]) generates the following flow dependence for the code in Figure 5.3:

$D_{flow} = \{ S1[j] \rightarrow S2[j, i] : (0 \leq j \leq n - 1) \land (j + 1 \leq i \leq n - 1) \}$

It characterizes the flow dependence that exists between the write reference $A[j][j]$ of statement $S1$ and the read reference $A[j][j]$ of statement $S2$: the value written by statement $S1$ at a particular iteration of the $j$ loop is used by statement $S2$ in the same iteration of the $j$ loop and in all iterations of the $i$ loop.

For our analysis, we consider exact dependences and exclude transitive dependences. That is, iterations involved in a dependence are such that, if the target iteration of a flow dependence reads a memory cell ‘$X$’, the corresponding source iteration is the last writer to the memory cell ‘$X$’ that precedes this reader.
5.4.2 Compile-time Use-count Determination

Affine loops are those where loop-bounds and array references are affine functions of loop iterators and program parameters. Affine computations form an important class of programs: the compute-intensive loops in many scientific codes (computational fluid dynamics, adaptive mesh refinement, numerical analysis etc.) are affine. According to a study [9], over 99% of loops in 7 out of the 12 programs of the SpecFP2000 and Perfect Club benchmarks are affine loops.

Algorithm 6 Compute flow dependence def_checksum contribution

Input: Flow Dependences : $D_{flow}$

Output: Statement and use-count pairs: $\rho$

1: for all Statements - $S_i$ do
2: $I_i \leftarrow$ Iteration Space of $S_i$
3: $I_i^{param} \leftarrow$ Parameterize all iterators of $I_i$
4: $Targets_i^{param} \leftarrow D_{flow}(I_i^{param})$
5: $use\_count^{param} \leftarrow |Targets_i^{param}|$
6: Add $\{S_i, use\_count^{param}\}$ to $\rho$
7: end for
8: return $\rho$

Algorithm 6 describes how we determine the number of uses, referred to as the use-count, for each definition. For each statement that produces a value, the definition statement, we determine the number of its consumers. The iterators of the surrounding loops of a statement under consideration are equated to some parameter values (line 3), $I_i^{param}$, allowing us to refer to a single parameterized iteration of that statement. In the flow dependences that map source iterations to their target iterations, $I_i^{param}$ is substituted as the source iteration and its target iterations are computed (line 4). The cardinality of the target
iteration set provides the definition statement’s use count (line 5). The algorithm returns a
collection of such statement and use-count pairs.

**Example.** The use-count for statement \( S_1 \) in the example code shown in Figure 5.3 is
computed as follows.

\[
I_{1}^{\text{param}} = \{ S_1[j] : j = jp \} \text{ where } jp \text{ is a parameter}
\]

\[
\text{Targets}_{1}^{\text{param}} = D_{\text{flow}}(I_{1}^{\text{param}}) = \{ S_2[jp,i] : (0 \leq jp \leq n - 1) \land (jp + 1 \leq i \leq n - 1) \}
\]

\[
|\text{Targets}_{1}^{\text{param}}| = \{ n - 1 - jp : (0 \leq jp \leq n - 2) \}
\]

The \( D_{\text{flow}} \) is the flow dependence shown in Section 5.4.1 for the example code. The value
written by the write reference \( A[j][j] \) of statement \( S_1 \) is used by the read reference \( A[j][j] \)
of statement \( S_2 \) in \( n-1-j \) following iterations (the lower bound of \( i \) loop is \( j+1 \), and the upper
bound is \( n-1 \), resulting in \( n-1-j \) iterations). Thus the use-count of \( S_1 \) is \( n-1-j \) as determined
by the algorithm. Further, it can be noticed that the use-count output above applies to
iterations of the \( j \) loop up to \( n-2 \) and the last iteration, when \( j \) is \( n-1 \), is excluded. This is
because the last iteration has no target iterations in statement \( S_2 \): when \( j=n-1 \), the lower
bound of the \( i \) loop (\( n \)) becomes greater than its upper bound (\( n-1 \)) resulting in no instance
of statement \( S_2 \) being executed when \( j=n-1 \).

The checksum-inserted version of the example code is shown in Figure 5.5. The
\texttt{add_to_chksm} macro definition takes three parameters: the checksum to add to (def/use
checksum), the value to add, and the number of times the value is to be added. The pro-
duced value is multiplied by the use-count and added to the def-checksum; consumed val-
ues are added to the use-checksum.
for $j = 0$ to $n-1$
    add_to_chksm(use Cs, $A[j][j]$, 1)

S1: $A[j][j] = \sqrt{A[j][j]}$
    if $j \leq n-2$
        add_to_chksm(def Cs, $A[j][j]$, $n-1-j$)
    for $i = j+1$ to $n-1$
        add_to_chksm(use Cs, $A[i][j]$, 1)
        add_to_chksm(use Cs, $A[j][j]$, 1)


Figure 5.5: Example affine code snippet with checksum augmentation

5.4.3 Optimization: Index-Set Splitting

In the checksum-augmented code shown in Figure 5.5, following statement S1, the error detection code that adds the value $A[j][j]$ written by statement S1 to the def-checksum has a branching structure: $A[j][j]$ is added to def-checksum $n-1-j$ times only for iterations of $j$ loop up to $n-2$.

To minimize performance penalty due to such control overheads, iteration space of the $j$ loop may be split so that in each of the split iteration spaces, $A[j][j]$ has the same use-count and thus, the need to evaluate if conditionals in each iteration of the $j$ loop is obviated. The loop-partitioned code thus formed is shown in Figure 5.6. The last iteration of the $j$ loop, when $j = n-1$, is peeled from the rest of the iterations. As explained above, statement S2 does not appear in the peeled iteration at $j=n-1$ due to the loop bounds.

Algorithm 7 describes the general procedure for splitting iteration spaces so that the use-count of a statement in a split iteration space remains the same for all iterations in that space. Inputs to the algorithm are the iteration spaces of all statements $I_{S_i}^{in}$, the index sets $\delta_{S_i}$ (such as $0 \leq j \leq n-2$, $j = n-1$ in the above example), and the schedule that defines the
for j = 0 to n−2 /*index split j*/
  add_to_chksm(use_cs,A[j][j],1)
S1:  A[j][j] = sqrt(A[j][j]);
  add_to_chksm(def_cs,A[j][j],n−1−j)
  for i = j+1 to n−1
    add_to_chksm(use_cs,A[i][j],1)
    add_to_chkksm(use_cs,A[j][j],1)
j = n−1 /*index split j*/
  add_to_chksm(use_cs,A[j][j],1)
  A[j][j] = sqrt(A[j][j])

Figure 5.6: Checksum-augmented example affine code with index-set splitting optimization

order of execution of the iteration spaces. The index sets $\delta_S$ act as the criteria according to which the iteration spaces are to be split.

For each iteration space, it is checked if a particular index set can cause the iteration space to be severed: if there are any common iterators between the index-set, and the iteration space then the index set potentially splits the iteration space (line 3). In the example code, any split of the $j$ loop affects iteration spaces of both statements $S1$ and $S2$. However, if only the $i$ loop is to be broken up, it does not affect iteration space of statement $S1$ as loop $i$ is not a surrounding loop for statement $S1$. For the common iterators identified, the iteration space is split so that the range of values an iterator assumes is a sub-range of values the same iterator assumes in the index set (line 4). This results in partitioning of the iteration space of that statement and ensures that no partition is a strict superset of the index set. The resultant smaller iteration spaces constitute $I_{S_j}^{out}$.

Any iteration points not yet a part of $I_{S_j}^{out}$ are subsequently added to $I_{S_j}^{out}$ so that all the iteration points contained in $I_{S_j}^{in}$ are included in $I_{S_j}^{out}$ (line 8). Finally, the code for the loop
Algorithm 7 Split iteration spaces

Input: Iteration Spaces : $I_{S_j}^{in}$, Index Sets : $\delta_{S_i}$, Schedules : $\theta$

Output: Loop Nest : $L$

1: for all Iteration Spaces : $I_{S_j}^{in}$ do
2:     for all Index Sets - $\delta_{S_i}$ do
3:         if Under $\theta$, $\text{Iterators}_{\text{common}} = \text{Iterators}_{\text{of}}\ \delta_{S_i} \cap \text{Iterators}_{\text{of}}\ I_{S_j} \neq \phi$ then
4:             $I_{S_j}^{out} \leftarrow \text{Split indexes of } I_{S_j}^{in} \text{ s.t } (\text{Range of } \text{Iterators}_{\text{common}} \in I_{S_j}^{in}) \subseteq (\text{Range of } \text{Iterators}_{\text{common}} \in \delta_{S_i})$
5:         end if
6:     end for
7: end for
8: $I_{S_j}^{out} \leftarrow I_{S_j}^{out} \cup (I_{S_j}^{in} \setminus I_{S_j}^{out})$
9: $L \leftarrow \text{Generate Code to traverse } I_{S_j}^{out} \text{ with schedule } \theta$
10: return $L$

nest is generated by traversing $I_{S_j}^{out}$ according to the schedule $\theta$ (line 9). The output of the algorithm is the index-set split loop nest.

5.5 Inspectors for Dynamic Start-time Use-count Determination

Hitherto, regular loops are examined whose iteration spaces and array accesses can be characterized and properties about them discerned at compile-time. Irregular codes in contrast require a combination of static and dynamic approaches to establish properties about them.

5.5.1 Basic Approach

Consider the code shown in Figure 5.7(a), that we use to illustrate the challenges and the solution approach for irregular codes. The variable $\text{temp}$ is defined, and then its uses are subject to $x[10]$ and $z[5]$ being non-zero. Therefore, depending on their values, $\text{temp}$ may be used once, twice, or not used at all. We proceed as follows. At the def-site, $\text{temp}$ is added to the def-checksum. At the use-site, $\text{temp}$ is added to use-checksum, and a counter
write temp

def_checksum += temp
e_def_checksum += temp

if (x[10]) {
    read temp
}

if (z[5]) {
    read temp
}

/* Epilogue */

def_checksum +=
temp*(temp_use_count-1);
e_use_checksum += temp;

assert(def_checksum == use_checksum)
assert(e_def_checksum == e_use_checksum)

(a) Original code

(b) Checksum-augmented code

Figure 5.7: Code snippet illustrating the general checksum-augmentation scheme

(temp_use_count) is incremented to keep track of the total number of times the variable gets used. Finally to match the checksums, the value of tmp is added to def-checksum, (temp_use_count − 1) times (it was added once to the def-checksum at the def-site). If there were no memory errors, then the two checksums match.

Doing only the above, it turns out, is not sufficient to catch all memory errors. Consider the scenario in which both the conditionals in the code shown evaluate to true, resulting in
two uses of temp. Let us further assume that the first read was correct. At this point, both def-checksum and use-checksum equal temp. Before the second read, let us suppose that a memory error occurred, and temp got changed to temp'. At second read, it gets added to use-checksum, and use-checksum is now equal to temp + temp'. And, at epilogue, temp' gets added to def-checksum (as temp_use_count would be 2), which makes def-checksum equal temp + temp', same as the present value of use-checksum, and the memory corruption goes undetected.

The problem with merely adding the current value of a variable to the def-checksum in the epilogue is that the corrupted value might get added to both the def- and use-checksums and thus escape detection. We fix the problem by defining auxiliary checksums: e_def_checksum and e_use_checksum. e_def_checksum is computed at the def-site as before, but e_use_checksum is computed only after the last use of the variable rather than at each use-site. The error detection code generated using the scheme described is shown in Figure 5.7(b). We note that the problem explained above now gets resolved: e_def_checksum = temp, but e_use_checksum = temp', and the memory error is detected. The modified scheme with the auxiliary checksums ensures that the value of the variable finally added to the def-checksum to match the number of its uses is not potentially corrupted.

5.5.2 Optimizations for Iterative Codes

Figure 5.8 shows an example of a code requiring dynamic support: accesses to array p_new at statement S1 are data-dependent. Further, number of iterations of the loop is dynamically determined. We describe how we generate error-detection checksums for the
while not converged
    for j1 = 0 to n-1
        S1: temp1 += p_new[cols[j1]]
        for j2 = 0 to n-1
            S2: temp2 += p_new[j2]
        for j3 = 0 to n-1
            S3: p_new[j3] = temp3
        /* convergence check not shown */

Figure 5.8: Example code requiring dynamic use count determination

element focusing on two arrays p_new, and cols followed by a description of the general scheme.

The following set of observations is made about the reads (uses) and writes (definitions) to array p_new:

- The reads to p_new at statement S1 are indexed by cols array entries. However, no element of array cols is written to in the while loop. Therefore, the same set of array elements of p_new is read in every iteration of the while loop.

- The elements of p_new read at statement S2 are amenable to compile-time analysis as the array index expressions and loop bounds of enclosed loop indexed by j_2 are affine.

- The new definition of array p_new at statement S3 is used a fixed number of times (even though not known at compile-time) before being overwritten in the next iteration of the while loop, if the algorithm has not converged.

An inspector is used to examine the cells of array p_new that will be read because of data-dependent accesses at statement S1. Since these reads are loop invariant, the inspector
is hoisted above the while loop to reduce the overhead of running the inspector. The parts of the code that are affine are subjected to static analysis techniques developed in the previous section, and the information from the inspector and static analysis are combined to generate the checksum calculation code.

The reads to array cols are affine; however, the number of iterations of the while loop is not known apriori. The number of accesses to a cell of array cols will be the number of accesses to it in an iteration of the loop, which can be determined by compile-time techniques, multiplied by the number of iterations. To determine the number of iterations of the loop, we introduce a new variable to count the number of dynamic executions of the loop. For such read references, the number of accesses is a function of the dynamic loop count. We observe that this count is not known at the time of the value’s definition. In such cases, we adopt the following method. At the definition site, the defined value is added to def-checksum and the auxiliary def-checksum e_def_checksum once. At each use site, the used value is added to use-checksum once. Post loop execution, in the epilogue code, the value is added to def-checksum one less than loop-count times. It is also added to the auxiliary use-checksum e_use_checksum to balance contributions to the def- and use-checksums.

Figure 5.9 shows the checksum-augmented version of the example in Figure 5.8. The inspector code counts the number of accesses to data-dependent references to array p_new. The loop is instrumented with def-checksum and use-checksum computation operations. Variable iter keeps count of the number of iterations of the while loop. In the epilogue code, the values of array cols are added to def-checksum iter − 1 times. The values of p_new are added to use-checksum in the epilogue as well to account for the fact that the new definition of p_new in the last iteration of the while loop goes unused.
/* Inspector */
for j1=0 to n-1
count_p_new[cols[j1]]++
iter = 0
while not converged
iter++
for j1=0 to n-1
   add_to_chksm(use_cs, cols[j1], 1)
   add_to_chksm(use_cs, p_new[cols[j1]], 1)
S1: temp1 += p_new[cols[j1]]
for j2=0 to n-1
   add_to_chksm(use_cs, p_new[j2], 1)
S2: temp2 += p_new[j2]
for j3=0 to n-1
S3: p_new[j3] = temp3
   add_to_chksm(def_cs, p_new[j3], count_p_new[j3]+1)
/* convergence check not shown */

/* Epilogue */
for i=0 to n-1
   add_to_chksm(def_cs, cols[i], iter-1)
   add_to_chkkm(e_use_cs, cols[i], 1)
   add_to_chkkm(use_cs, p_new[i], count_p_new[i]+1)

Figure 5.9: Example code with inspector to determine use counts and the generalized checksum-augmentation scheme

5.6 Overall Approach To Transient Error Detection

We employ one global checksum to track all definitions of values and another global checksum to track all uses. A mismatch between the two checksums indicates the occurrence of one or more data corruption errors. In general, array accesses are classified into two categories: affine and non-affine. Affine references are analyzed using compile-time techniques developed in Section 5.4. For non-affine accesses, inspector code is generated to examine number of uses of any array cell. The inspector is hoisted out of the looping
structure when legally feasible, when the indexing data structures are not modified in the loop. The number of uses of a definition from affine and non-affine uses are combined to arrive at the use-count for a definition.

Algorithm 8 Insert error detection checksums

**Input:** The abstract syntax tree of a program: \( AST \)

**Output:** The AST of equivalent resilient program: \( AST' \)

1. \( \text{Live} - in \leftarrow \text{Gather live-in values and associated use counts in the AST} \)
2. \( \text{Prologue} \leftarrow \text{Generate operations adding } \text{Live} - in \text{ to } \text{def checksum} \text{ and to auxiliary } e_{\text{def checksum}} \)
3. \( \text{for all } \text{Use Site} \in AST \text{ do} \)
4. \( \text{for all } \text{Def Site} \in AST \text{ do} \)
5. \( \text{if } \text{use count} \text{ is known then} \)
6. \( \text{end if} \)
7. \( \text{end for} \)
8. \( \text{end for} \)

19: \( \text{Adjustment Pending Defs} \leftarrow \text{Gather variables in the } \text{AST whose use count was not known} \)
20: \( \text{for all } \text{Var} \in \text{Adjustment Pending Defs} \text{ do} \)
21: \( \text{Insert code in Epilogue to add Var to } \text{def checksum, use count} - 1 \text{ times} \)
22: \( \text{end for} \)
23: \( \text{Verifier} \leftarrow \text{Add code to assert equality of def and use checksums} \)
24: \( \text{AST}' \leftarrow \text{Append } \{\text{Prologue,AST,Epilogue,Verifier}\} \)
Algorithm 8 presents the general approach to generating the error detection checksums. At each use-site, the read values are added to the use-checksum. The use-counts are incremented if the reads are in irregular accesses. At a def-site, if the number of the uses can be predetermined (such as described in Section 5.4 for affine array references), then the produced value is added to def-checksum as many times as it will be used. If the number of uses for a definition of a value cannot be determined a priori, the newly defined value is added once to def-checksum and once to an auxiliary def-checksum, to be adjusted in the epilogue code. A use-count is also maintained and initialized to zero. However, before the new value is written, checksum adjustments are made for the previous value of that variable about to be overwritten: the old value is added to def-checksum, use_count−1 times, and is also added to update auxiliary use-checksum. At the end of the program, the verifier code that compares the def- and use-checksums is introduced to detect any memory errors that might have occurred to any variables during execution of the program.

Any commutative and associative checksum operator can be chosen for error detection. We use integer modulo addition as the checksum function in this work. If an operand is a floating-point number or a character, its bit representation is viewed as an integer of the appropriate size before invoking the checksum operation. Another candidate checksum operator that has associative and commutative properties is exclusive or (XOR). We use integer arithmetic because it is supported as efficiently in hardware as XOR while exhibiting superior fault coverage [78].

The fault model considered in this work (Section 5.2.2) assumes that the ALU and registers are resilient to errors and the memory subsystem that includes caches, memory, write queues is vulnerable to errors. The checksum approach requires that the produced values be resident in registers until they are added to def-checksum and conversely, the consumed
values that are added to use-checksum reside in registers until their actual use. Load and store operations due to register spills require additional checksum contributions to ensure their correctness. The checksums need to reside in registers in order that they themselves are not subject to memory errors. Bit-flips in the checksums could result in fault positives: an error maybe flagged even when the data are not corrupted. Overflows in checksum values do not affect the presented scheme as integer modulo addition is associative and commutative. An inherent property of checksums is that they can trigger false negatives: errors canceling each other out and resulting in a correct checksum value. We empirically evaluate the probabilities of fault negatives of the checksum scheme in Section 5.7.1.

The scheme can cause a false positive (i.e., checksums do not match even when program correctness is not affected) if, some uses of a definition occur in data-dependent paths and none are taken: When the use-count of a definition is not known a priori then the produced value is added to def-checksum. If the use-count of such a definition remains 0 (that is, the produced value is not consumed), but a bit-flip has affected it, then it will cause def- and use- checksums to not match. However, this does not affect program results as the corrupted value is not used.

We now discuss the correctness of the checksum scheme.

**Theorem 5.6.1.** The checksum computation codes inserted by Algorithm 8 correctly detect all memory errors with high probability provided checksums are register-resident.

**Proof.** We want to prove that checksums flag an error – def- and use-checksums do not match – if for any variable, the value assigned to the variable at its definition-site is different from the values carried by the variable at any of its subsequent use-sites. We also want to show that various checksum adjustments are correct and do not trigger false positives: checksums match if there are no memory errors.
We first consider the case where there is a single variable in the program. Let the value assigned to the variable at def-site be \( v \) and it be used \( n \) times. Now we can have two cases: 1) use-count, the number of uses of the variable, is known at compile-time or 2) uses occur in data-dependent paths of the AST and use-count is not known at compile-time.

**Case 1 (known use-count).** When use-count, say \( n \), is known at compile-time, value \( v \) is multiplied by \( n \) and is added to def-checksum. Therefore, def-checksum is \( n \times v \). We analyze what happens to checksums for different values of \( n \) and in the presence of errors.

(a) \( n = 0 \): Both def-checksum and use-checksum are 0. Since number of uses is zero, data errors do not happen.

(b) \( n = 1 \): At the use-site, if the value read is \( v \), the two checksums match, otherwise the error is caught.

(c) \( n > 1 \): Let the values read during the \( n \) reads be \( v_1, v_2, \ldots, v_n \). The checksum values will be def_checksum = \( n \times v \) and use_checksum = \( v_1 + v_2 + \cdots + v_n \). If there were no memory errors then each \( v_i \) will be equal to \( v \) and the two checksums will match. If any of the reads are wrong, then with a high probability def-checksum and use-checksum will not match and the error will be detected. A false negative can happen when \( n \times v = v_1 + v_2 + \cdots + v_n \) even though some \( v_i \)'s are different from \( v \).

**Case 2 (unknown use-count).** At the def-site, value \( v \) is used to initialize def-checksums: def_checksum = \( v \) and e_def_checksum = \( v \).

(a) \( n = 0 \): Since there are no uses, use-checksum and use-count are 0. In the epilogue code or just before the variable is overwritten, value \( v \) is added to the def-checksum
use\_count − 1 (= −1) times. Therefore, def-checksum becomes 0. v is added to auxiliary checksum: e\_use\_checksum = v. Thus, def\_checksum = use\_checksum = 0 and e\_def\_checksum = e\_use\_checksum = v unless there are memory errors.

(b) \( n = 1 \): When there is only one use, def-checksum is not adjusted since use\_count − 1 = 0 and memory errors are detected with certainty.

(c) \( n > 1 \): Let the values read during the \( n \) reads be: \( v_1, v_2, \ldots, v_n \). Therefore, use\_checksum = \( v_1 + v_2 + \cdots + v_n \). When adjusting def-checksum, let the value of the variable be \( v_{n+1} \).

Hence, finally def\_checksum = \( v + (n - 1) \times v_{n+1} \) and
e\_use\_checksum = \( v_{n+1} \). If there are no errors, then the checksums match: def\_checksum = use\_checksum = \( n \times v \) and
e\_def\_checksum = e\_use\_checksum = v. When there is an error, it is caught if either \( v + (n - 1) \times v_{n+1} \neq v_1 + v_2 + \cdots + v_n \) or \( v \neq v_{n+1} \).

Use of e\_checksums prevents an important class of errors going undetected: If after first use, \( v \) changes to \( v' \) and the error persists, then def\_checksum = use\_checksum = \( v + (n - 1) \times v' \). However, auxiliary checksums correctly flag the error: e\_def\_checksum = v and e\_use\_checksum = \( v' \) and they do not match.

Given that the checksum operator is commutative and associative, an analysis similar to the one above can be employed for multiple variables to establish that the checksum values match if there are no errors and they do not match with a high probability in the presence of memory errors.
5.7 Experimental Evaluation

We first evaluate the fault coverage achieved by the checksum scheme and then characterize the performance overheads incurred due to the checksum computations.

5.7.1 Effectiveness of Checksums

A characteristic of the use of checksums for error detection is that errors in multiple values contributing to a checksum could cancel one another and produce a seemingly correct checksum even though there were bit flips in the data. We therefore empirically assess the percentage of errors that can escape detection. One-bit errors are always caught. We therefore inject multi-bit errors into an array of 64-bit integers and monitor the percentage of cases in which checksums are successful in detecting them. Over 100,000 trials, the following steps are repeated: The data are initialized and initial 64-bit checksum is computed. Then, either two, three, four, five, or six bits over all bits of the array are uniformly randomly selected and values of those bits are flipped. A 64-bit checksum is again computed and is compared with the initial checksum. If there is a mismatch in the checksum values, then the injected error has been correctly caught, otherwise error has escaped detection. The percentage instances when errors are not caught over 100,000 trials is reported. For small array sizes, we gathered data for even higher number of trials, up to 1 million trials. The percentage of undetected errors we observed was very similar to what we report here for 100,000 trials.

Effectiveness of Checksums

Table 5.1 shows the percentage of cases in which a multi-bit error (2, 3, 4, 5, 6 bit-flips) is not caught for arrays with $10^2, 10^4, 10^6$ 64-bit integers under the column header “One
Table 5.1: Percentage of undetected errors with integer modulo addition checksums

<table>
<thead>
<tr>
<th>#bit-flips</th>
<th>N</th>
<th>One checksum</th>
<th>Two checksums</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All 0 bits</td>
<td>All 1 bits</td>
<td>Random bits</td>
</tr>
<tr>
<td>2</td>
<td>$10^2$</td>
<td>0.025%</td>
<td>0.025%</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>0.014%</td>
<td>0.014%</td>
</tr>
<tr>
<td></td>
<td>$10^6$</td>
<td>0.014%</td>
<td>0.014%</td>
</tr>
<tr>
<td>3</td>
<td>$10^2$</td>
<td>0.002%</td>
<td>0.002%</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>0.002%</td>
<td>0.002%</td>
</tr>
<tr>
<td></td>
<td>$10^6$</td>
<td>0.002%</td>
<td>0.002%</td>
</tr>
<tr>
<td>4</td>
<td>$10^2$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>$10^6$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>5</td>
<td>$10^2$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>$10^6$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>6</td>
<td>$10^2$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>$10^4$</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td></td>
<td>$10^6$</td>
<td>0%</td>
<td>0%</td>
</tr>
</tbody>
</table>

checksum”. Experiments are carried out over three kinds of initial data: all bits of the array elements are set to 0; all bits are set to 1; bits are randomly initialized.

Two-bit errors experience the highest percentage of undetected errors. The number of undetected errors approaches zero as the number of bits that are flipped increases. With multiple bit flips (greater than 2), even though a pair of errors in different values might line up to cancel each other out, other bit flips may not exactly line up and the error is still caught with a high probability.

Among different types of data values, the percentage of undetected errors is highest for random values. When all bits of all array elements are initialized to either zeroes or ones
and bits at the same bit position in two array elements flip, the next significant bit is also changed and the error will be caught unless the bit in the next significant position is also flipped. In contrast, random data are likely to have an equal number of zeroes and ones, and when a 0 becomes 1, and a 1 becomes 0 at the same bit-position in two values, it will not change the carry bit and thus the error can go undetected. Further, we observe that the performance of checksums is largely independent of the number of values that go into forming the checksum. In over 99% of cases, errors are successfully detected.

Additional complementary evaluations of checksum-effectiveness are presented in addenda 5.A and 5.B.

**Use of Multiple Checksums**

The odds of error detection can be further improved and protection against data corruption fortified with the use of multiple checksums: the first checksum is a direct sum of the values as before. The second checksum is obtained by adding permutations of the values encountered.

We evaluate the fault coverage when two checksums are used: two 64-bit checksums are computed from initial data, errors are injected, and checksums are recomputed. If there is a mismatch between either pair of checksum values, the error has been detected. While forming the second checksum, each array element is left-rotated by an amount determined by five bits of the address of that array element, viz., $4^{th}$ to $8^{th}$ least significant bits. The address of a 64-bit integer, which occupies 8 bytes, will likely be a multiple of 8. Hence, the first three least significant bits, $1^{st}$ to $3^{rd}$, are likely to be zero. Thus, each value that goes into computation of the second checksum is left-rotated by an amount between 0 and 31 ($2^5 - 1$) bits depending on its address.
Therefore, with a high probability the corrupted values are rotated by different amounts, which prevents the lined-up erroneous bits in one checksum from lining up to cancel one another out after rotation in the other checksum as well. The percentages of undetected errors using the two checksum scheme are shown in Table 5.1 under the column header “Two checksums”. Only a very small percentage of two-bit errors are undetected, and all three-, four-, five-, six-bit errors are detected.

5.7.2 Performance Overheads

Table 5.2 lists the benchmark programs and problem sizes used to measure performance overheads of the checksum-augmented codes. All array references in ADI, cholesky, dsyrk, jacob1d, LU, seidel, strmm, and trisolv are statically analyzable, whereas a subset of array references in CG, and moldyn are irregular and for those references, dynamic analysis techniques are applied. The affine benchmark codes are taken from the PLUTO benchmark suite [92].

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Problem size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADI</td>
<td>Alternating direction implicit solver</td>
<td>TSteps = 500, N = 3000</td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate gradient</td>
<td>TSteps = 1500, NZ = 513072</td>
</tr>
<tr>
<td>cholesky</td>
<td>Cholesky decomposition</td>
<td>N = 3000</td>
</tr>
<tr>
<td>dsyrk</td>
<td>Symmetric rank-k update</td>
<td>N = 3000</td>
</tr>
<tr>
<td>jacob1d</td>
<td>1-D Jacobi stencil computation</td>
<td>TSteps = 100000, N = 400000</td>
</tr>
<tr>
<td>LU</td>
<td>LU decomposition</td>
<td>N = 3000</td>
</tr>
<tr>
<td>moldyn</td>
<td>Molecular dynamics</td>
<td>TSteps = 100000, N = 400000</td>
</tr>
<tr>
<td>seidel</td>
<td>2-D seidel stencil</td>
<td>TSteps = 500, N = 3000</td>
</tr>
<tr>
<td>strsm</td>
<td>Triangular matrix equations solver</td>
<td>N = 3000</td>
</tr>
<tr>
<td>trisolv</td>
<td>Triangular system of linear equations solver</td>
<td>N = 3000</td>
</tr>
</tbody>
</table>
The performance overheads are evaluated on an Intel Xeon E5630 processor running at 2.53GHz with 32KB L1 cache. The programs are compiled using Intel icc 13.1.3 compiler, with -O3 optimization flag. Each benchmark is run five times and the average execution time is reported.

**One Checksum Software-only Solution**

![Normalized running time of the resilient codes with one checksum software-only solution. Absolute running times, in seconds, of the Original unmodified codes are also shown](image)

Figure 5.10: Normalized running time of the resilient codes with one checksum software-only solution. Absolute running times, in seconds, of the Original unmodified codes are also shown

Figure 5.10 shows the performance of the checksum-augmented programs, referred to as resilient versions, as compared to the performance of the original non-resilient versions. The geometric mean of performance overheads of resilient codes across all benchmarks is 78.8% (the Resilient bars in the figure). When the index-set splitting optimization presented in Section 5.4.3 and inspector-hoisting optimization described in Section 5.5.2 are applied,
performance overheads are reduced to 40.2% on average (geometric mean) (the Resilient-Optimized bars).

The index-set splitting transformation partitions a loop so that statements in a partitioned loop have the same use count. This also removes any conditional statements introduced in the loop on account of the use count of a definition changing based on the values of a loop iterator. Index-set splitting thus reduces control overheads and improves performance. Further, it enables vectorization if the compiler was not able to vectorize a loop because of conditional statements in the loop. For example, the original LU code is vectorized, with a running time of 11.1 seconds, while its resilient version is not vectorized, incurring a running time is 30.3 seconds. When index-set splitting transformation is applied to the resilient code, the compiler successfully vectorizes the resulting loops, and the resulting running time is 13.2 seconds.

The highest overhead is observed for the moldyn benchmark. While moldyn is an iterative code, the inspector for one of the arrays used cannot be hoisted out as loop-invariant properties are not preserved. Therefore, counters are used to keep track of the number of uses of array elements. The CG benchmark is an iterative method that includes sparse matrix-vector multiplications. Even though it has irregular accesses, each iteration of the computation has the same data access pattern. Hence, the inspector code for running through irregular accesses is hoisted out of the iterative loop. The optimized running time shown in Figure 5.10 for CG includes the effect of both index-set splitting and inspector-hoisting optimizations. However, all performance benefits for the CG benchmark accrue from the inspector-hoisting optimization: running time increases from 33.7s for the original version to 81.1s for the resilient version, but reduces to 52.7s for the optimized resilient version.
Evaluating Overheads under Hardware Support

Software-based checksum schemes computing one checksum incur a non-trivial overhead. Tracking multiple checksums in software would be too expensive to be used in practice. Here, we evaluate the potential improvements in the costs when hardware assistance is available. Rather than realize the optimal hardware design to support def-use checksums, we present a candidate design and estimate the likely performance overheads.

We consider the presence of special checksum functional units to compute the checksums. We assume sufficient parallelism in these units so that they never lead to stalls in the execution pipeline. For example, one checksum unit could be associated with every functional unit to compute the checksums on the inputs and outputs of that functional unit. Given that a checksum unit only performs integer modulo arithmetic, the hardware space requirements would be similar to that of an integer ALU, and much lesser than that of a floating point unit. Note that a more detailed analysis of the execution pipeline might achieve the same effect with less resources dedicated to checksum computation.

The intermediate values of the checksums are stored in internal registers and are reduced to a scalar when requested. We assume the presence of instructions to set and query the checksums. In addition, we assume instructions to specify the checksum contribution behavior of another instruction. For example, a checksum instruction could specify which of the operands of a following floating-point instruction contribute to the def- and use-checksums. In addition, def-checksums can be associated with a scale factor based on the use count. Use counts that are small constants are encoded directly in the checksum instruction. For large or non-constant use counts, we assume the presence of a use-count register that is set by a separate instruction. A checksum instruction can scale the value...
defined by a subsequent arithmetic or memory instruction by a small constant or by the
value in the use-count register.

We estimate the cost of supporting checksums by transforming the index-split resilient
version of each benchmark to exploit such a feature set. Because the checksums can now
be computed in hardware, we remove the operations that compute them in software. In
place of the actual checksum calculation operations, we insert instructions corresponding
to the checksum instructions. For all benchmarks other than CG and moldyn, one check-
sum instruction is inserted to precede every floating point arithmetic operation. For CG
and moldyn, which also checksum the integer indirection arrays, we insert a checksum in-
struction for every memory operation. Note that these are more instructions than required.
However they provide adequate coverage for all checksum points of interest.

![Graph showing estimated performance of resilient codes](image)

**Figure 5.11:** Estimated performance of resilient codes with a special function unit to
compute checksums. y-axis employs a linear scale with selective tick marks to highlight
the costs.
We considered candidate instructions to estimate the overhead and report results for a nop instruction (no operation—the instruction is fetched and decoded but does not use any functional unit resources) is used to represent each checksum instruction. The nop instruction is of sufficient width to pack the actions specified by the checksum instruction. Because insertion of assembly instructions might interfere with compiler optimizations, the nop instructions are inserted into optimized assembly generated by the compiler (icc −O3 in this case). The operations to compute the use counts and all operations in the prologue and epilogue portions of the resilient code are still retained.

Note that this design can support multiple checksums without introducing any additional software overhead. We evaluate the overhead of this design by benchmarking the resulting nop-inserted code. The resulting overheads are shown in Figure 5.11. We observe that the overheads are significantly reduced, with the largest overheads (4%–10%) incurred by moldyn, seidel, and trisolv. We observed that the input original version of trisolv was performing worse than the index-split version and made the index-split version, without any checksums, the baseline for comparison in both Figure 5.10 and Figure 5.11. We observe a speedup in strsm due to the compiler vectorization having different effects on the two versions of the code. Excluding strsm, we observe an average overhead of 3% (geometric mean), demonstrating that hardware support could help significantly reduce the overheads in computing the checksums while also supporting multiple checksums. um functional units to compute the checksums. We assume sufficient parallelism in these units so that they never lead to stalls in the execution pipeline. For example, one checksum unit could be associated with every functional unit to compute the checksums on the inputs and outputs of that functional unit. Given that a checksum unit only performs integer modulo arithmetic, the hardware space requirements would be similar to that of an integer ALU,
and much lesser than that of a floating point unit. Note that a more detailed analysis of the execution pipeline might achieve the same effect with less resources dedicated to checksum computation.

The intermediate values of the checksums are stored in internal registers and are reduced to a scalar when requested. We assume the presence of instructions to set and query the checksums. In addition, we assume instructions to specify the checksum contribution behavior of another instruction. For example, a checksum instruction could specify which of the operands of a following floating-point instruction contribute to the def- and use-checksums. In addition, def-checksums can be associated with a scale factor based on the use count. Use counts that are small constants are encoded directly in the checksum instruction. For large or non-constant use counts, we assume the presence of a use-count register that is set by a separate instruction. A checksum instruction can scale the value defined by a subsequent arithmetic or memory instruction by a small constant or by the value in the use-count register.

We estimate the cost of supporting checksums by transforming the index-split resilient version of each benchmark to exploit such a feature set. Because the checksums can now be computed in hardware, we remove the operations that compute them in software. In place of the actual checksum calculation operations, we insert instructions corresponding to the checksum instructions. For all benchmarks other than CG and moldyn, one checksum instruction is inserted to precede every floating point arithmetic operation. For CG and moldyn, which also checksum the integer indirection arrays, we insert a checksum instruction for every memory operation. Note that these are more instructions than required. However they provide adequate coverage for all checksum points of interest.
We considered candidate instructions to estimate the overhead and report results for a nop instruction (no operation—the instruction is fetched and decoded but does not use any functional unit resources) is used to represent each checksum instruction. The nop instruction is of sufficient width to pack the actions specified by the checksum instruction. Because insertion of assembly instructions might interfere with compiler optimizations, the nop instructions are inserted into optimized assembly generated by the compiler (icc −O3 in this case). The operations to compute the use counts and all operations in the prologue and epilogue portions of the resilient code are still retained.

Note that this design can support multiple checksums without introducing any additional software overhead. We evaluate the overhead of this design by benchmarking the resulting nop-inserted code. The resulting overheads are shown in Figure 5.11. We observe that the overheads are significantly reduced, with the largest overheads (4%–10%) incurred by moldyn, seidel, and trisolv. We observed that the input original version of trisolv was performing worse than the index-split version and made the index-split version, without any checksums, the baseline for comparison in both Figure 5.10 and Figure 5.11. We observe a speedup in strsm due to the compiler vectorization having different effects on the two versions of the code. Excluding strsm, we observe an average overhead of 3% (geometric mean), demonstrating that hardware support could help significantly reduce the overheads in computing the checksums while also supporting multiple checksums.

5.8 Related Work

Fault Tolerance. Approaches to tackling soft errors have been considered at various levels of the hardware-software environment and typically involve redundancy coupled with periodic validation. At the lowest level, hardware checkers such as self-checking logic [83]
and hardware duplication [7] provide the most general coverage, but can be expensive in terms of chip area, performance, and power, and not widely available on all systems of interest.

More general schemes on commodity hardware resort to different forms of execution redundancy and periodic validation, possibly aided with micro-architecture support. This includes approaches for simultaneous multithreading [48] and chip multiprocessors [102] that employ redundantly executing threads whose results are periodically compared. Process-level redundancy [107] involves duplicating the inputs and entire execution on distinct processes whose outputs are then compared. These schemes incur significant overheads or require specialized hardware to frequently validate the ongoing computation.

Software-level redundancy techniques that are agnostic of application structure duplicate instructions within a single thread of execution and introduce additional checking instructions for validation. Such an approach could check the computation [87] or control flow [86], while duplicating all application state. SWIFT [100] checks the computation and control state without duplicating application state and assumes that memory is made fault-tolerant through other means such as ECC. The approach presented in this chapter can complement SWIFT by decoupling correctness checks for the memory subsystem from that for control and computation.

An alternative to redundancy-based techniques, symptom-based detection techniques employ low-cost detectors that observe violation of application visible properties, such as loop trip counts and invariants [54], as the outcome of underlying hardware faults. These solutions can be software-only [120, 47] or combine hardware support [91] in the design of low-cost symptom-based detectors. These incur lower overheads as compared to redundancy-based techniques while potentially trading off fault coverage. Hari et al. [55]
observe the trade-off between fault detection latency – the delay in detecting a fault – and the detection overhead in symptom-based detectors. These techniques focus on detection of errors in computation or control flow instructions. Our work focuses on the design of symptom-based detectors for multi-bit memory errors.

Schroeder et al. [104] observed that the likelihood of repeated failures in DRAM increases after a first failure has occurred. Multi-bit memory errors have been observed in SRAM soft error evaluation experiments [76, 89]. Yoon et al. [130, 131] designed approaches to virtualize ECC for main memory so as to increase its flexibility and offload expensive ECC error correction for last-level caches to DRAM. Gold et al. [46] observe that multi-bit error detection and correction in L1 caches is more expensive in terms of performance, power, and area even for reasonable sizes.

Shirvani et al. [106] designed approaches to provide checksum protection by periodically scrubbing memory, rather than check every read and write operation, which lowers fault coverage as compared to our approach. Checksum approaches for various data structures such as trees have been considered by exploiting structure-specific properties [17]. Algorithm-based fault tolerance for linear algebra relies on distributivity of floating point multiplication over addition to make specific array operations resilient [59]. Chen et al. [24] optimize the management of checksums for such algorithm-based fault tolerance schemes. Blum et al. [12] presented checkers that can validate operations on data structures stored in unreliable memory using the minimal amount of reliable memory required. Our approach does not rely on any assumptions about floating point arithmetic and is not restricted to specific algorithms or data structures.

The Flikker system [73] distinguishes critical and non-critical data in programs and stores non-critical data in DRAM memories that are refreshed at lower rates thereby saving
energy but introducing data corruptions. The compiler-assisted approach in this chapter can complement Flikker and similar systems in detecting errors in data stored in unreliable memory, further enabling the use of such energy saving strategies.

Maxino [78] evaluates error detection effectiveness of different checksum algorithms, namely, exclusive or, two’s complement addition, one’s complement addition, Fletcher checksum, Adler checksum, and Cyclic Redundancy Codes.

Compile-time Analysis and Transformation. Griebl et al. [50] use the index-set splitting approach to form regular dependence structures so that effective loop transformations can be applied. As there can be a large number of ways of splitting loops, they address the problem of efficiently finding index sets that yield good loop transformations. The index-set splitting procedure developed in this work, on the other hand, addresses the problem of systematically achieving separation of iteration spaces according to a given criterion: the criterion in this work being that the same use count be applicable to writes in each split index set. Inspector-executor strategies have been used in prior work to perform start-time optimizations to exploit data structure and dependence properties not known at compile-time [95].

5.9 Conclusion

The decreasing transistor sizes, use of lower voltage levels, and smaller noise margins have increased the probability of multi-bit errors in the memory subsystem. Therefore, it is of increased interest to design efficient solutions to address this problem in software as hardware does not typically have embedded multi-bit error detection and correction mechanisms. To this end, we have developed novel compiler techniques to instrument application programs with error detection codes that protect every memory reference at runtime. The
experimental evaluation demonstrates that the proposed solutions have low overheads and are practical.
Addendum

A significant portion of the ideas and experiments presented in this addendum is contributed by Paul Sivilotti [108].

5.A An empirical checksum evaluation independent of array size

The fault model is that, high energy particle strikes could affect any bits including the same bits repeatedly. Therefore, undetected error-rate analysis can be carried out by reasoning about bits in which position get affected and with what probability and such an analysis will be independent of the number of words involved in forming of the checksums. In other words, the undetected error-rate will be the same irrespective of the number of data-words.

We describe a set of experiments to determine undetected error-rates for data with different distributions of 0 and 1 bits. The data-word and checksum are 64-bits in length. When a 0-bit in the $i^{th}$ position is flipped, it will increase the checksum value by $2^i$ modulo $2^{64}$. Similarly, if a 1-bit in the $i^{th}$ position is changed to 0, it will decrease the checksum by $2^i$ modulo $2^{64}$. When a total of $k$ bits are flipped, a correct checksum will be produced only when the combined effect (in terms of increase and decrease in checksum value) will be such that, the checksum value remains unchanged.
Table 5.3: Number of undetected errors with integer modulo addition checksums in 1 million trials

<table>
<thead>
<tr>
<th>#bit-flips</th>
<th>0.0000</th>
<th>0.0625</th>
<th>0.1250</th>
<th>0.1875</th>
<th>0.2500</th>
<th>0.3125</th>
<th>0.3750</th>
<th>0.4375</th>
<th>0.5000</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>246</td>
<td>2015</td>
<td>3553</td>
<td>4913</td>
<td>5954</td>
<td>6937</td>
<td>7340</td>
<td>7948</td>
<td>8068</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>50</td>
<td>94</td>
<td>99</td>
<td>157</td>
<td>135</td>
<td>187</td>
<td>174</td>
<td>185</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>15</td>
<td>37</td>
<td>93</td>
<td>104</td>
<td>145</td>
<td>166</td>
<td>178</td>
<td>198</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>11</td>
<td>15</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

Length of prefix of 0’s is varied from 64 to 0

<table>
<thead>
<tr>
<th>#bit-flips</th>
<th>64</th>
<th>56</th>
<th>48</th>
<th>40</th>
<th>32</th>
<th>24</th>
<th>16</th>
<th>8</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>246</td>
<td>1253</td>
<td>2170</td>
<td>3149</td>
<td>4184</td>
<td>5169</td>
<td>6098</td>
<td>7033</td>
<td>8068</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>47</td>
<td>66</td>
<td>75</td>
<td>96</td>
<td>126</td>
<td>157</td>
<td>187</td>
<td>185</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>3</td>
<td>22</td>
<td>42</td>
<td>38</td>
<td>82</td>
<td>111</td>
<td>148</td>
<td>198</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Methodology. Over 1 million trials, $k$ bit-flips are injected into the 64-bit data-word. The number of instances in those 1 million trials when the checksum value remains unchanged represents occurrences of false negatives.

To simulate $k$ bit-flips, in each trial, $k$ bits are selected one after another (a particular bit may be selected more than once). Each bit in the 64-bit word is chosen with equal probability of $\frac{1}{64}$.

Data Patterns. The selected bit could be 0 or 1 depending on the data patterns being simulated, which are described below.
• **Probability of a bit being 1:** In the 64-bit data word, any of the 64 bits being 1 is varied from 0.0000 through 0.5000. When the probability is 0.0000, each time a bit is selected, it is considered 0. When the probability is 0.0625, the selected bit is 1 with probability 0.0625: a random number between 0 and 1 is generated and if the generated random number is less than 0.0625, the bit is considered 1, otherwise it is treated as 0. When the probability is 0.5000, the selected bit is 0 or 1 with equal probabilities.

We note that, analysis for undetected errors when probability of a bit being 1 exceeds 0.5, let us say, $p (p > 0.5)$, the undetected error-rate will be identical to the case when the probability is $1 - p$. This is because, probability $p$ of a bit being 1 corresponds to the case when probability of a bit being 0 is $1 - p$ and the number of undetected errors occurring in a given bit-pattern and its flipped pattern (where all bits are flipped) are the same.

• **Length of prefix 0’s:** In the 64-bit word, the leading bits (most significant bits) are set to 0 and the rest of the trailing bits are random – a trailing bit is a 0 or a 1 with probability 0.5. The prefix length of leading 0 bits is varied from 0 through 64. When the 0-prefix length is 0, all bits are random and when it is 64, all bits are always 0.

Table 5.3 reports the number of undetected errors for different numbers of bit-flips — 2, 3, 4, 5, 6, 7 bit-flips. We observe that as the probability of occurrence of a 1 increases (as the data pattern becomes more random), the number of undetected errors also increases. The reason for increased undetected error-rate is that, if the bit-flips in a bit-position affect different bits (i.e., a 0 and a 1 bit), that does not change the carry bit, and therefore, has an increased chance of producing a correct checksum value.
5.B Checksum evaluation for benchmarks

We now measure undetected error-rate in data consumed by benchmark programs. The fractions of consumed values having 1 in each of the 64 bit positions is first calculated by running benchmark applications. Then, for the data pattern thus obtained, memory errors are injected and the number of undetected errors over 1 million trials is assessed.

Table 5.4 and 5.5 report the probability of occurrence of 1 in all 64 bit positions for three different benchmarks – dsyrk, CG, and LU. The data patterns closely resemble random data.
Table 5.4: Probability of occurrence of 1 in first 32 bit positions

<table>
<thead>
<tr>
<th>bit-position</th>
<th>dsyrk</th>
<th>CG</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.490562</td>
<td>0.485399</td>
<td>0.347111</td>
</tr>
<tr>
<td>1</td>
<td>0.494988</td>
<td>0.496783</td>
<td>0.432081</td>
</tr>
<tr>
<td>2</td>
<td>0.501221</td>
<td>0.497512</td>
<td>0.457986</td>
</tr>
<tr>
<td>3</td>
<td>0.493982</td>
<td>0.501835</td>
<td>0.474723</td>
</tr>
<tr>
<td>4</td>
<td>0.495747</td>
<td>0.501123</td>
<td>0.485075</td>
</tr>
<tr>
<td>5</td>
<td>0.489923</td>
<td>0.497725</td>
<td>0.491769</td>
</tr>
<tr>
<td>6</td>
<td>0.494331</td>
<td>0.497915</td>
<td>0.495248</td>
</tr>
<tr>
<td>7</td>
<td>0.495869</td>
<td>0.496383</td>
<td>0.497290</td>
</tr>
<tr>
<td>8</td>
<td>0.494595</td>
<td>0.504931</td>
<td>0.498378</td>
</tr>
<tr>
<td>9</td>
<td>0.496540</td>
<td>0.502417</td>
<td>0.498882</td>
</tr>
<tr>
<td>10</td>
<td>0.495637</td>
<td>0.501428</td>
<td>0.499419</td>
</tr>
<tr>
<td>11</td>
<td>0.493134</td>
<td>0.495659</td>
<td>0.499313</td>
</tr>
<tr>
<td>12</td>
<td>0.502876</td>
<td>0.500799</td>
<td>0.499621</td>
</tr>
<tr>
<td>13</td>
<td>0.503061</td>
<td>0.491189</td>
<td>0.499408</td>
</tr>
<tr>
<td>14</td>
<td>0.503238</td>
<td>0.470148</td>
<td>0.499444</td>
</tr>
<tr>
<td>15</td>
<td>0.502882</td>
<td>0.456634</td>
<td>0.499855</td>
</tr>
<tr>
<td>16</td>
<td>0.500521</td>
<td>0.452493</td>
<td>0.499655</td>
</tr>
<tr>
<td>17</td>
<td>0.496803</td>
<td>0.396256</td>
<td>0.499924</td>
</tr>
<tr>
<td>18</td>
<td>0.497023</td>
<td>0.387432</td>
<td>0.499438</td>
</tr>
<tr>
<td>19</td>
<td>0.501495</td>
<td>0.379595</td>
<td>0.499492</td>
</tr>
<tr>
<td>20</td>
<td>0.497018</td>
<td>0.370290</td>
<td>0.499463</td>
</tr>
<tr>
<td>21</td>
<td>0.500010</td>
<td>0.365520</td>
<td>0.499590</td>
</tr>
<tr>
<td>22</td>
<td>0.500937</td>
<td>0.366076</td>
<td>0.499580</td>
</tr>
<tr>
<td>23</td>
<td>0.495543</td>
<td>0.366682</td>
<td>0.499653</td>
</tr>
<tr>
<td>24</td>
<td>0.495771</td>
<td>0.360904</td>
<td>0.499549</td>
</tr>
<tr>
<td>25</td>
<td>0.498239</td>
<td>0.363192</td>
<td>0.499740</td>
</tr>
<tr>
<td>26</td>
<td>0.492085</td>
<td>0.367816</td>
<td>0.499634</td>
</tr>
<tr>
<td>27</td>
<td>0.490764</td>
<td>0.367178</td>
<td>0.499774</td>
</tr>
<tr>
<td>28</td>
<td>0.492199</td>
<td>0.360032</td>
<td>0.499918</td>
</tr>
<tr>
<td>29</td>
<td>0.488957</td>
<td>0.361758</td>
<td>0.499787</td>
</tr>
<tr>
<td>30</td>
<td>0.489108</td>
<td>0.364141</td>
<td>0.499651</td>
</tr>
<tr>
<td>31</td>
<td>0.490752</td>
<td>0.365960</td>
<td>0.499718</td>
</tr>
</tbody>
</table>
Table 5.5: Probability of occurrence of 1 in last 32 bit positions

<table>
<thead>
<tr>
<th>bit-position</th>
<th>dsyrk</th>
<th>CG</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.490177</td>
<td>0.356384</td>
<td>0.499813</td>
</tr>
<tr>
<td>33</td>
<td>0.495708</td>
<td>0.362954</td>
<td>0.499596</td>
</tr>
<tr>
<td>34</td>
<td>0.493416</td>
<td>0.360689</td>
<td>0.499432</td>
</tr>
<tr>
<td>35</td>
<td>0.500277</td>
<td>0.364852</td>
<td>0.499509</td>
</tr>
<tr>
<td>36</td>
<td>0.491688</td>
<td>0.365798</td>
<td>0.499705</td>
</tr>
<tr>
<td>37</td>
<td>0.496409</td>
<td>0.365445</td>
<td>0.499759</td>
</tr>
<tr>
<td>38</td>
<td>0.498835</td>
<td>0.356974</td>
<td>0.499620</td>
</tr>
<tr>
<td>39</td>
<td>0.491035</td>
<td>0.363342</td>
<td>0.499513</td>
</tr>
<tr>
<td>40</td>
<td>0.497682</td>
<td>0.366055</td>
<td>0.499724</td>
</tr>
<tr>
<td>41</td>
<td>0.500153</td>
<td>0.371747</td>
<td>0.499632</td>
</tr>
<tr>
<td>42</td>
<td>0.494113</td>
<td>0.364254</td>
<td>0.499702</td>
</tr>
<tr>
<td>43</td>
<td>0.498026</td>
<td>0.366738</td>
<td>0.499461</td>
</tr>
<tr>
<td>44</td>
<td>0.494156</td>
<td>0.367487</td>
<td>0.499055</td>
</tr>
<tr>
<td>45</td>
<td>0.496545</td>
<td>0.358428</td>
<td>0.498430</td>
</tr>
<tr>
<td>46</td>
<td>0.492784</td>
<td>0.364034</td>
<td>0.497538</td>
</tr>
<tr>
<td>47</td>
<td>0.491015</td>
<td>0.361461</td>
<td>0.494234</td>
</tr>
<tr>
<td>48</td>
<td>0.484306</td>
<td>0.361275</td>
<td>0.488841</td>
</tr>
<tr>
<td>49</td>
<td>0.473519</td>
<td>0.345241</td>
<td>0.477650</td>
</tr>
<tr>
<td>50</td>
<td>0.451221</td>
<td>0.339879</td>
<td>0.455583</td>
</tr>
<tr>
<td>51</td>
<td>0.408623</td>
<td>0.300999</td>
<td>0.415008</td>
</tr>
<tr>
<td>52</td>
<td>0.492307</td>
<td>0.362294</td>
<td>0.499361</td>
</tr>
<tr>
<td>53</td>
<td>0.550171</td>
<td>0.369526</td>
<td>0.509480</td>
</tr>
<tr>
<td>54</td>
<td>0.675266</td>
<td>0.403506</td>
<td>0.538704</td>
</tr>
<tr>
<td>55</td>
<td>0.175984</td>
<td>0.377156</td>
<td>0.837290</td>
</tr>
<tr>
<td>56</td>
<td>0.259332</td>
<td>0.353559</td>
<td>0.250035</td>
</tr>
<tr>
<td>57</td>
<td>0.008283</td>
<td>0.254505</td>
<td>0.226193</td>
</tr>
<tr>
<td>58</td>
<td>0.008283</td>
<td>0.256098</td>
<td>0.226193</td>
</tr>
<tr>
<td>59</td>
<td>0.008283</td>
<td>0.256568</td>
<td>0.226193</td>
</tr>
<tr>
<td>60</td>
<td>0.008283</td>
<td>0.256568</td>
<td>0.226193</td>
</tr>
<tr>
<td>61</td>
<td>0.008283</td>
<td>0.256568</td>
<td>0.226193</td>
</tr>
<tr>
<td>62</td>
<td>0.982855</td>
<td>0.471913</td>
<td>0.773807</td>
</tr>
<tr>
<td>63</td>
<td>0.000000</td>
<td>0.433673</td>
<td>0.476528</td>
</tr>
</tbody>
</table>
Table 5.6: Number of undetected errors with integer modulo addition checksums in 1 million trials

<table>
<thead>
<tr>
<th>#bit-flips</th>
<th>dsyrk</th>
<th>CG</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6945</td>
<td>7443</td>
<td>7781</td>
</tr>
<tr>
<td>3</td>
<td>182</td>
<td>162</td>
<td>202</td>
</tr>
<tr>
<td>4</td>
<td>162</td>
<td>161</td>
<td>184</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.6 shows the number of undetected errors for 2-, 3-, 4-, 5, 6-, and 7-bit flips. The number of false-negatives for benchmark data is comparable to number of occurrence of undetected errors when the probability of a bit being 1 is between 0.3125 and 0.4375 (Table 5.6).
Chapter 6: Future Work

Despite advances made in the areas of Parallel Programming and Computing, they remain grand challenges to solve for the computer science community. I am interested in investigating ways to find right abstractions to write parallel programs in; to optimize execution of programs for both performance, and energy metrics; to develop compiler technology to automatically convert legacy sequential codes to efficient parallel software; to propose techniques to easily debug/correct parallel codes.

**Code Synthesis for Cloud Applications** Cloud computing is emerging as a cost effective, and easy-to-use computing paradigm for businesses, and individuals. Yet, writing programs that take full advantage of the vast computing, and storage resources available on the Cloud is a challenge for most programmers for several factors: one has to write parallel programs, there may be multiple ways of writing parallel programs with differing communication costs, topology of inter-connects in the underlying parallel system plays an important role in determining the best assignment of tasks to processors etc. Therefore, compiler technology to relieve the programmer from the concerns of specifics of the execution environments will be developed.

From a given sequential program or a high level specification, techniques to automatically generate cloud applications will be designed. For regular applications, abstractions
of the Polyhedral model will be leveraged to generate code for cloud applications in frameworks such as, *MapReduce*. For irregular codes, domain specific languages will be developed and high performance code will be generated in the cloud programmable languages.

**Automatic Algorithm Discovery and Free Scheduling** The current compiler and dynamic analysis techniques rely on a given program text, or program representation, or execution to perform analyses to find optimizations for codes such as tilability, parallelizability, or vectorizability. A limitation of the reliance on a given program structure is that it may fail to uncover feasibility of a desired code transformation not due to intrinsic constraints imposed by the algorithm the program implements, but as an artifact of the way the program implements the algorithm. For example, none of the existing compiler techniques can determine that the Floyd-Warshall code that finds shortest paths in a graph is tilable. Nonetheless, researchers have proposed ingenious methods to apply tiling program transformation to the Floyd-Warshall code, that is semantically equivalent to the original program.

A possible way to explore all feasible code restructurings is to convert the given program to a form where it represents the semantic core of the algorithm: outputs of the program are functions of input invariables alone (and do not involve any temporary variables). Once we obtain the semantic core, then we can explore the best ways to schedule operations of the algorithm according to a given metric such as performance, or energy, or a combination of both. As to finding all feasible restructurings, mathematical properties of the operations performed are taken into consideration such as, associativity of \( \min \) and \( \max \) operations, symmetric, associative properties of addition operations.
Compiler-Directed Designation of Synchronization Points  Software errors (bugs) in parallel programs are hard to pin-point, and fix. Parallel programs suffer from a greater variety of bugs than sequential programs: data races, improper placement of synchronization primitives are peculiar to parallel codes. Further, if one defines critical sections, and synchronization points conservatively (i.e., where strictly not necessary), that will lead to much degraded performance. Past research has developed techniques to identify parallel bugs, but solutions to automatically fix those errors are lacking.

I would like to devise mechanisms to correctly, and efficiently derive requisite synchronizations (defining of critical sections, designation of atomic regions, placement of barriers). The idea is to solicit a specification and its parallel implementation (without any synchronization constructs defined in it) and then explore different points in the code to place synchronizations so that the parallel execution results in the same output as the specification. However, there may be a large number of candidate synchronization points. Ways to efficiently search through the large space of possible synchronization placements that result in correct execution, and best performance will be investigated.
Chapter 7: Conclusion

The technology trends present myriad challenges for compilers — how to generate parallel programs that are high performing on a diverse set of parallel architectures, the growing power-wall problem makes it critical to develop compiler support to reduce energy expenditure in various hardware components, increasing rates of transient hardware failures force development of software-level techniques to deal with soft-errors.

This dissertation presents compiler techniques to tackle above mentioned challenges. It proposes adaptive tiling methods to achieve high performance, a compile-time analysis to determine effective cache size to reduce leakage energy in cache memories, compiler support for software cache coherence, and a compiler-pass to insert checksum computation codes in application programs to make them resilient against memory errors.
Bibliography


