Cache Optimizations for Stream Programs

by

Jānis Sermuliņš

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2005

© Massachusetts Institute of Technology 2005. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author . . . .
Department of Electrical Engineering and Computer Science
May 6, 2005

Certified by . . . .
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Cache Optimizations for Stream Programs

by

Jānis Sermuliņš

Submitted to the Department of Electrical Engineering and Computer Science
on May 6, 2005, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

As processor speeds continue to increase, the memory bottleneck remains a primary impediment to attaining performance. Effective use of the memory hierarchy can result in significant performance gains. This thesis focuses on a set of transformations that either reduce cache-miss rate or reduce the number of memory accesses for the class of streaming applications, which are becoming increasingly prevalent in embedded, desktop and high-performance processing. A fully automated optimization algorithm is presented that reduces the memory bottleneck for stream applications developed in the high-level stream programming language StreamIt.

This thesis presents four memory optimizations: 1) cache aware fusion, which combines adjacent program components while respecting instruction and data cache constraints, 2) execution scaling, which judiciously repeats execution of program components to improve instruction and state locality, 3) scalar replacement, which converts certain data buffers into a sequence of scalar variables that can be register allocated, and 4) optimized buffer management, which reduces the overall number of memory accesses issued by the program. The cache aware fusion and execution scaling reduce the instruction and data cache-miss rates and are founded upon a simple and intuitive cache model that quantifies the temporal locality for a sequence of actor executions. The scalar replacement and optimized buffer management reduce the number of memory accesses.

An experimental evaluation of the memory optimizations is presented for three different architectures: StrongARM 1110, Pentium 3 and Itanium 2. Compared to unoptimized StreamIt code, the memory optimizations presented in this thesis yield a 257% speedup on the StrongARM, a 154% speedup on the Pentium 3, and a 152% speedup on Itanium 2. These numbers represent averages over our streaming benchmark suite. The most impressive speedups are demonstrated on an embedded processor StrongARM, which has only a single data and a single instruction cache, thus increasing the overall cost of memory operations and cache misses.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I would like to thank William Thies and Rodric Rabbah for their guidance throughout my work that led to this thesis. This thesis is an expanded version of a paper [24] by the author, William Thies, Rodric Rabbah and Saman Amarasinghe that will appear in proceedings of ACM SIGPLAN/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems. I would also like to thank all members of the StreamIt group. I would like to thank William Thies for his work on filter fusion and loop unrolling in the StreamIt compiler. I would like to thank Jasper Lin for his work on scalar replacement and loop unrolling in the StreamIt compiler. I would also like to thank my advisor, Saman Amarasinghe, for his guidance.

Most importantly, I would like to thank my parents.

Contents

1 Introduction **13**

1.1 Overview 14

1.2 Organization 16

2 Background **17**

2.1 StreamIt 17

2.1.1 Hierarchical Streams 17

2.1.2 Execution Model 19

2.1.3 Compilation Process 20

2.1.4 Implementation of Cache Optimizations 21

3 Cache Model **23**

3.1 Instruction Cache 24

3.2 Data Cache 28

4 Optimization Algorithm **31**

4.1 Cache Optimizations 32

4.1.1 Cache Aware Fusion 32

4.1.2 Execution Scaling 35

4.2 Scalar Replacement 37

4.2.1 Scalar Replacement Example 37

4.2.2 Implications for the Cache Aware Fusion 38

4.2.3 Implications for Unrolling 39

4.3 Optimized Buffering of Live Items 40
4.3.1 Modulation 41
4.3.2 Copy-Shift 42
4.3.3 Optimized Copy-Shift 43

5 Experimental Evaluation **45**
5.1 Evaluation of Cache Aware Fusion, Scaling and Scalar Replacement . 47
5.2 Evaluation of Copy-Shift and Modulation 50
5.3 Evaluation of Peek-Scaling and Cut-Peek 53
5.4 Comparison to Cache Unaware Full Fusion 56
5.5 Evaluation of Modified Cache Aware Fusion for Pentium 3 and Itanium 2 58

6 Related Work **61**

7 Conclusion **65**
7.1 Future Work 66

A Experimental Evaluation of Execution Scaling Heuristic **67**

List of Figures

2-1 StreamIt code for an FIR filter 18

2-2 Hierarchical streams in StreamIt. 18

2-3 Example pipeline with FIR filter. 19

2-4 Example pipeline. 20

2-5 C code for running the steady state 20

3-1 Impact of execution scaling on performance. 27

4-1 Outline of the cache aware fusion algorithm 34

4-2 Our heuristic for calculating the scaling factor. 35

4-3 Example StreamIt code 37

4-4 Generated C code corresponding to the fused filter with no unrolling . 38

4-5 Generated C code corresponding to the fused filter with full unrolling and scalar replacement 38

5-1 Impact on average execution time for our benchmark suite. 47

5-2 Performance results for StrongARM 1110 49

5-3 Performance results for Pentium 3 49

5-4 Performance results for Itanium 2 49

5-5 Original StreamIt code for the buffer test. 51

5-6 Performance of buffer management strategies on a StrongARM 1110 . 52

5-7 Performance of buffer management strategies on a Pentium 3 52

5-8 Performance of buffer management strategies on an Itanium 2 52

5-9 Performance of peek-scaling and cut-peek on a StrongARM 1110 . . . 55

5-10 Performance of peek-scaling and cut-peek on a Pentium 3 55
5-11 Performance of peek-scaling and cut-peek on an Itanium 2 55
5-12 Comparison to full fusion on a StrongARM 1110 57
5-13 Comparison to full fusion on a Pentium 3 57
5-14 Comparison to full fusion on an Itanium 2 57
5-15 Performance of cache optimizations after instruction limit modification on a Pentium 3 59
5-16 Performance of cache optimizations after instruction limit modification on an Itanium 2 59

List of Tables

5.1 Evaluation benchmark suite. 46
5.2 The best performing buffer management strategies for each benchmark-architecture pair [along with speedup over CAF+scaling+SR] 53

Chapter 1

Introduction

As processor speeds continue to increase, the memory bottleneck remains a primary impediment to attaining performance. Effective use of the memory hierarchy can result in significant performance gains. Current practices for hiding memory latency are invariably expensive and complex. For example, superscalar processors resort to out-of-order execution to hide the latency of cache misses. This results in large power expenditures and also increases the cost of the system. Compilers have also employed computation and data reordering to improve locality, but this requires a heroic analysis due to the obscured parallelism and communication patterns in traditional languages such as C.

For performance-critical programs, the complexity inevitably propagates all the way to the application developer. Programs are written to explicitly manage parallelism and to reorder the computation so that the instruction and data working sets fit within the cache. For example, the inputs and outputs of a procedure might be arrays that are specifically designed to fit within the data cache on a given architecture; loop bodies are written at a level of granularity that matches the instruction cache. While manual tuning can be effective, the end solutions are not portable. They are also exceedingly difficult to understand, modify, and debug.

The recent emergence of streaming applications presents an opportunity to mitigate these problems using simple transformations in the compiler. Stream programs are rich with parallelism and regular communication patterns that can be exploited by the compiler to automatically tune memory performance. Streaming codes encompass a broad spectrum of applications, including embedded communications processing, multimedia encoding and playback, compression, and encryption. They also range to server applications, such as HDTV editing and hyper-spectral imaging. It is natural to express a stream program as a high-level graph of independent components, or *actors*. Actors communicate using explicit FIFO channels and can execute whenever a sufficient number of data items are available on their input channels. In a stream graph, actors can be freely combined and reordered to improve caching behavior as long as there are sufficient inputs to complete each execution. Such transformations can serve to automate tedious approaches that are performed manually using today's languages; they are too complex to perform automatically in hardware or in the most aggressive of C compilers.

1.1 Overview

A naïve way to execute a stream program on a uniprocessor is to execute all program components in some precomputed order. However, the size of the instruction footprint of the whole program may not fit into the instruction cache. Thus we need to divide the stream program into parts such that each part has an instruction footprint that fits into the instruction cache; we then scale the execution of the parts to amortize the instruction and data cache misses associated with loading the instructions and state variables associated with each part into the instruction and data cache. The execution scaling needs to be judicious so that the data produced by a scaled stream program part does not exceed the data cache.

This thesis presents four memory optimizations for stream programs: *(i)* cache aware fusion, *(ii)* execution scaling, *(iii)* scalar replacement, and *(iv)* optimized buffer management. This thesis also presents a simple quantitative model of caching behavior for streaming workloads, providing a foundation to reason about the transformations that improve cache usage. Work in this thesis is done in the context of the Synchronous Dataflow [18] model of computation, in which each actor in the stream graph has a known input and output rate. This is a popular model for a broad range of signal processing and embedded applications.

Cache aware fusion combines adjacent actors into a single function. This allows the compiler to optimize across actor boundaries. The fusion algorithm presented in this thesis is cache aware in that it never fuses a pair of actors that will result in an overflow of the data or the instruction cache. However, our experimental evaluation will show that on some architectures we can relax the instruction cache constraint to allow more aggressive optimization across actor boundaries.

Execution scaling is a transformation that improves instruction locality by executing each fused actor in the stream graph multiple times before moving on to the next actor. Since an actor that has been produced using cache aware fusion usually fits within the cache, the repeated executions serve to amortize the cost of loading the actors instruction stream and state from off-chip memory. However, as the cache model will show, actors should not be scaled excessively, as their outputs will eventually overflow the data cache. This thesis presents a simple and effective algorithm for calculating a scaling factor that respects both instruction and data constraints. The cache aware fusion in conjunction with execution scaling represent a *unified approach* that simultaneously considers the instruction and data working sets.

As actors are fused together, new buffer management strategies become possible. The most aggressive of these, termed scalar replacement, serves to replace an array with a series of local scalar variables. Unlike array references, scalar variables can be register allocated, leading to large performance gains.

We also present several optimized buffer management strategies for FIFO channels that always have to retain a set of live items. We compare two implementations: using a circular buffer, and periodically shifting the live items to the start of the buffer. Our experimental evaluation suggests that shifting the live items is the best implementation if the shifting is performed infrequently.

The memory optimizations presented in this thesis are implemented as part of StreamIt, a language and compiler infrastructure for stream programming [27]. We evaluate the optimizations on three architectures. The StrongARM 1110 represents an embedded system without a secondary cache, the Pentium 3 represents a superscalar processor and the Itanium 2 represents a VLIW processor. We find that cache aware fusion, scalar replacement, execution scaling and optimized buffer management each offer significant performance gains, and the most consistent speedups result when all are applied together. Compared to unoptimized StreamIt code, the optimizations presented in this thesis yield a 257% speedup on the StrongARM, a 154% speedup on the Pentium 3, and a 152% speedup on Itanium 2. These numbers represent averages over our streaming benchmark suite.

1.2 Organization

This thesis is organized as follows. Chapter 2 gives background information on the StreamIt language. Chapter 3 lays the foundation for the cache optimizations by presenting a quantitative model of caching behavior for any sequence of actor executions. Chapter 4 describes cache aware fusion, execution scaling, scalar replacement and optimized buffer management in detail. Chapter 5 evaluates optimizations proposed in this thesis as they were implemented in the StreamIt compiler. Finally, Chapter 6 describes related work and Chapter 7 concludes the thesis.

Chapter 2

Background

In this chapter we present StreamIt, a high level stream programming language [27].

2.1 StreamIt

StreamIt is an architecture independent language that is designed for stream programming. In StreamIt, programs are represented as graphs where nodes represent computation and edges represent FIFO-ordered communication of data over tapes. See [27], [10], [28], [17] and [14] for more information and research about StreamIt.

2.1.1 Hierarchical Streams

In StreamIt, the basic programmable unit (i.e., an actor) is a *filter*. Each filter contains a special function (called `work` function) that executes atomically, popping (i.e., reading) a fixed number of items from the filter's input tape and pushing (i.e., writing) a fixed number of items to the filter's output tape. A filter may also `peek` at a given index on its input tape without consuming the item; this makes it simple to represent computation over a sliding window. The `push`, `pop`, and `peek` rates are declared as part of the work function, thereby enabling the compiler to construct a static schedule of filter executions. An example implementation of a Finite Impulse Response (FIR) filter appears in Figure 2-1.

```
float->float filter FIRFilter (int N, float[] weights) {
  // declare work function along with I/O rates
  work push 1 pop 1 peek N {
    float sum = 0;
    for (int i = 0; i < N; i++) {
      // examine items on the input queue
      sum += peek(i) * weights[i];
    }
    pop(); // remove an item from the input queue
    push(sum); // enqueue the sum onto the output queue
  }
}
```

Figure 2-1: StreamIt code for an FIR filter

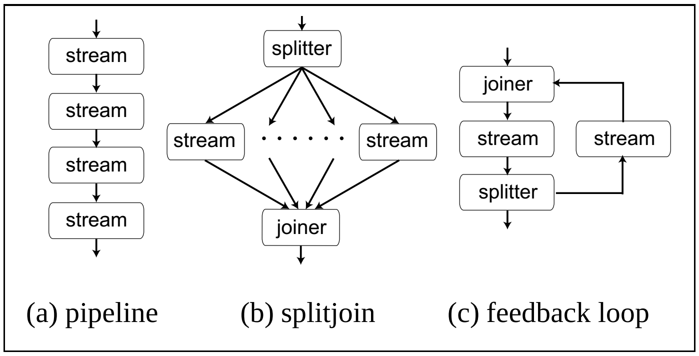
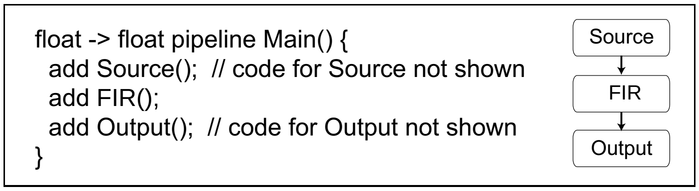


Figure 2-2: Hierarchical streams in StreamIt.

The work function is invoked (fired) whenever there is sufficient data on the input tape. For the FIR example in Figure 2-1, the filter requires at least `N` elements before it can execute. The value of `N` is known at compile time when the filter is constructed. A filter is akin to a class in object oriented programming with the work function serving as the main method. The parameters to a filter (e.g., `N` and `weights`) are equivalent to parameters passed to a class constructor.

In StreamIt, the application developer focuses on the hierarchical assembly of the stream graph and its communication topology, rather than on the explicit management of the data buffers between filters. StreamIt provides three hierarchical structures for composing filters into larger stream graphs (see Figure 2-2). The *pipeline* construct composes streams in sequence, with the output of one connected to the input of the next. An example of a pipeline appears in Figure 2-3.



```
float -> float pipeline Main() {
  add Source();  // code for Source not shown
  add FIR();
  add Output();  // code for Output not shown
}
```

Figure 2-3: Example pipeline with FIR filter.

The *splitjoin* construct distributes data to a set of parallel streams, which are then joined together in a roundrobin fashion. In a splitjoin, the *splitter* performs the data scattering, and the *joiner* performs the gathering. A splitter is a specialized filter with a single input and multiple output channels. On every execution step, it can distribute its output to any one of its children in either a *duplicate* or a *roundrobin* manner. For the former, incoming data are replicated to every sibling connected to the splitter. For the latter, data are scattered in a roundrobin manner, with each item sent to exactly one child stream, in order. The splitter type and the weights for distributing data to child streams are declared as part of the syntax (e.g., `split duplicate` or `split roundrobin(`$w_1, \ldots, w_n$`))`. The splitter counterpart is the joiner. It is a specialized filter with multiple input channels but only one output channel. The joiner gathers data from its predecessors in a roundrobin manner (declared as part of the syntax) to produce a single output stream.

StreamIt also provides a *feedback loop* construct for introducing cycles in the graph.

2.1.2 Execution Model

As noted earlier, an actor (i.e., a filter, splitter, or joiner) executes whenever there are enough data items on its input tape. In StreamIt, actors have two epochs of execution: one for initialization, and one for the *steady state*. The initialization primes the input tapes to allow filters with peeking (i.e. *peek rate* > *pop rate*) to execute the very first instance of their work functions. A steady state is an execution that does not change the buffering in the channels: the number of items on each channel after the execution is the same as it was before the execution. Every valid stream graph has a steady state [18], and within a steady state, there are often many possibilities for interleaving

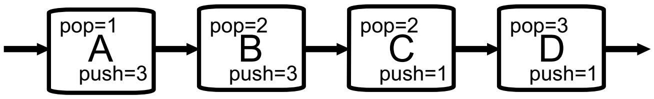


Figure 2-4: Example pipeline.

```
run_steady_state() {
  A_work(4); // execute Filter A 4 times
  B_work(6); // execute Filter B 6 times
  C_work(9); // execute Filter C 9 times
  D_work(3); // execute Filter D 3 times
}
```

Figure 2-5: C code for running the steady state

actor executions. An example of a steady state for the pipeline in Figure 2-4 requires filter `A` to fire 4 times, `B` 6 times, `C` 9 times, and `D` 3 times.

2.1.3 Compilation Process

The StreamIt compiler derives the initialization and steady state schedules [15] and outputs a C program that includes the initialization and work functions, as well as a driver to execute each of the two schedules. The compilation process allows the StreamIt compiler to focus on high level optimizations, and relies on existing compilers to perform machine-specific optimizations such as register allocation, instruction scheduling, and code generation—this two-step approach affords us a great deal of portability (e.g., code generated from the StreamIt compiler is compiled and run on three different machines as reported in Chapter 5).

For example, referring to Figure 2-4, the compiler generates C code for running the steady state that is shown in Figure 2-5.

To execute the program, the steady state is wrapped with another loop that invokes the steady state a designated number of times. Preceding the steady state, a similar initialization schedule is run to prime the data buffers.

2.1.4 Implementation of Cache Optimizations

The cache optimization algorithm presented in this thesis and described in more detail in the Chapter 4 has been implemented in the StreamIt optimizing stream compiler. The cache optimization algorithm first uses cache aware fusion to combine adjacent actors such that each fused actor can fit its instruction and data footprint within the instruction and data cache. The cache optimization algorithm then optimizes fused actors by performing aggressive loop unrolling, scalar replacement, constant propagation and other optimizations supported by the StreamIt compiler. A special compiler pass has been implemented by the author that creates the top level function that invokes the work functions of granularity adjusted actors, scales their execution and implements optimized buffer management strategy.

Chapter 3

Cache Model

From a caching point of view, it is intuitively clear that once an actor's instruction working set is fetched into the cache, we can maximize instruction locality by running the actor as many times as possible. This of course assumes that the total code size for all actors in the steady state exceeds the capacity of the instruction cache. For the benchmarks used in this thesis, the total code size for a steady state ranges from 2 Kb to over 135 Kb (and commonly exceeds 16 Kb). Thus, while individual actors may have a small instruction footprint, the total footprint of the actors in a steady state exceeds a typical instruction cache size. From these observations, it is evident that we must *scale* the execution of actors in the steady state in order to improve temporal locality. In other words, rather than running a actor n times per steady state, we scale it to run $m \times n$ times. We term m the *scaling factor*.

The obvious question is: to what extent can we scale the execution of actors in the steady state? The answer is non-trivial because scaling, while beneficial to the instruction cache behavior, may overburden the data cache as the buffers between actors may grow to prohibitively large sizes that degrade the data cache behavior. Specifically, if a buffer overflows the cache, then producer-consumer locality is lost.

This chapter presents a simple and intuitive cache model to estimate the instruction and data cache miss rates for a steady state sequence of actor firings. The model serves as a foundation for reasoning about the cache aware optimizations introduced in this thesis. We develop the model first for the instruction cache, and then generalize it to account for the data cache.

3.1 Instruction Cache

A steady state execution is a sequence of actor firings $S = (a_1, \ldots, a_n)$, and a *program execution* corresponds to one or more repetitions of the steady state. We use the notation $S[i]$ to refer to the actor a that is fired at logical time i, and $|S|$ to denote the length of the sequence.

Our cache model is simple in that it considers each actor in the steady state sequence, and determines whether one or more misses are bound to occur. The miss determination is based on the the *instruction reuse distance* (*IRD*), which is equal to the number of unique instructions that are referenced between two executions of the actor under consideration (as they appear in the schedule). The steady state is a compact representation of the whole program execution, and thus, we simply account for the misses within a steady state, and generalize the result to the whole program. Within a steady state, an actor is charged a miss penalty if and only if the number of referenced instructions since the last execution (of the same actor) is greater than the instruction cache capacity.

Formally, let $\mathit{phase}(S, i)$ for $1 \leq i \leq |S|$ represent a subsequence of k elements of S:

$$\mathit{phase}(S, i) = (S[i], S[i+1], \ldots, S[i+k-1])$$

where $k \in [1, |S|]$ is the smallest integer such that $S[i+k] = S[i]$. In other words, a phase is a subsequence of S that starts with the specified actor ($S[i]$) and ends before the next occurrence of the same actor (i.e., there are no intervening occurrences of $S[i]$ in the phase). Note that because the steady state execution is cyclic, the construction of the subsequence is allowed to wrap around the steady state[1]. For example, the steady state $S_1 = (\mathtt{AABB})$ has $\mathit{phase}(S_1, 1) = (\mathtt{A})$, $\mathit{phase}(S_1, 2) = (\mathtt{ABB})$, $\mathit{phase}(S_1, 3) = (\mathtt{B})$, and $\mathit{phase}(S_1, 4) = (\mathtt{BAA})$,

---

[1] In other words, the subsequence is formed from a new sequence $S' = S|S$ where $|$ represents concatenation.

Let $I(a)$ denote the code size of the work function for actor a. Then the instruction reuse distance is

$$\mathit{IRD}(S, i) = \sum_{a} I(a)$$

where the sum is over all distinct actors a occurring in $\mathit{phase}(S, i)$. We can then determine if a specific actor will result in an instruction cache miss (on its next firing) by evaluating the following step function:

$$\mathit{IMISS}(S, i) = \begin{cases} 0 & \text{if } \mathit{IRD}(S, i) \leq C_I\text{; hit: no cache refill,} \\ 1 & \text{otherwise; miss: (some) cache refill.} \end{cases} \tag{3.1}$$

In the equation, C_I represents the instruction cache size.

Using Equation 3.1, we can estimate the instruction miss rate (IMR) of a steady state as:

$$\mathit{IMR}(S) = \frac{1}{|S|} \sum_{i=1}^{|S|} \mathit{IMISS}(S, i). \tag{3.2}$$

Our cache model allows us to rank the quality of an execution ordering: schedules that boost temporal locality result in miss rates closer to zero, and schedules that do not exploit temporal locality result in miss rates closer to one.

For example, in the steady state $S_1 = (\texttt{AABB})$, assume that the combined instruction working sets exceed the instruction cache, i.e., $I(\texttt{A}) + I(\texttt{B}) > C_I$. Then, we expect to suffer a miss at the start of every steady state because the phase that precedes the execution of `A` (at $S_1[1]$) is $\mathit{phase}(S_1, 2)$ with an instruction reuse distance greater than the cache size ($\mathit{IRD}(S_1, 2) > C_I$). Similarly, there is a miss predicted for the first occurrence of actor `B` since $\mathit{phase}(S_1, 4) = (\texttt{BAA})$ and $\mathit{IRD}(S_1, 4) > C_I$. Thus, $\mathit{IMR}(S_1) = 2/4$ whereas for the following variant $S_2 = (\texttt{ABAB})$, $\mathit{IMR}(S_2) = 1$. In the case of S_2, we know that since the combined instruction working sets of the actors exceed the cache size, when actor `B` is fired following `A`, it evicts part of actor `A`'s instruction working set. Hence when we transition back to fire actor `A`, we have to refetch certain instructions, but in the process, we replace parts of actor `B`'s working set. In terms of the cache model, $IRD(S_2, i) > C_I$ for every actor in the sequence, i.e., $1 \leq i \leq |S_2|$.

Note that the amount of refill is proportional to the number of cache lines that are replaced when swapping actors, and as such, we may wish to adjust the cache miss step function (*IMISS*). One simple variation is to allow for some partial replacement without unduly penalizing the overall value of the metric. Namely, we can allow the constant C_I to be some fraction greater than the actual cache size. Alternatively, we can use a more complicated miss function with a more uniform probability distribution.

Temporal Locality According to our model, the concept of improving temporal instruction locality translates to deriving a steady state where, in the best case, every actor has only one phase that is longer than unit-length. For example, a permutation of the actors in S_2 (where all phases are of length two) that improves temporal locality will result in S_1, which we have shown has a relatively lower miss rate.

Execution Scaling Another approach to improving temporal locality is to scale the execution of the actors in the steady state. Scaling increases the number of consecutive firings of the same actor. A scaled steady state has a greater number of unit-length phases (i.e., a phase of length one and the shortest possible reuse distance).

We represent a scaled execution of the steady state as $S^m = (a_1^m, \ldots, a_n^m)$: the steady state S is scaled by m, which translates to $m - 1$ additional firings of every actor. For example, scaling $S_1 = (\mathtt{AABB})$ by a factor of two results in $S_1^2 = (\mathtt{AAAABBBB})$ and scaling $S_2 = (\mathtt{ABAB})$ by the same amount results in $S_2^2 = (\mathtt{AABBAABB})$;

From Equation 3.1, we observe that unit-length phases do not increase the instruction miss rate as long as the size of the actor's instruction working set is smaller than the cache size; we assume this is always the case. Therefore, scaling has the effect of preserving the pattern of miss occurrences while also lengthening the steady

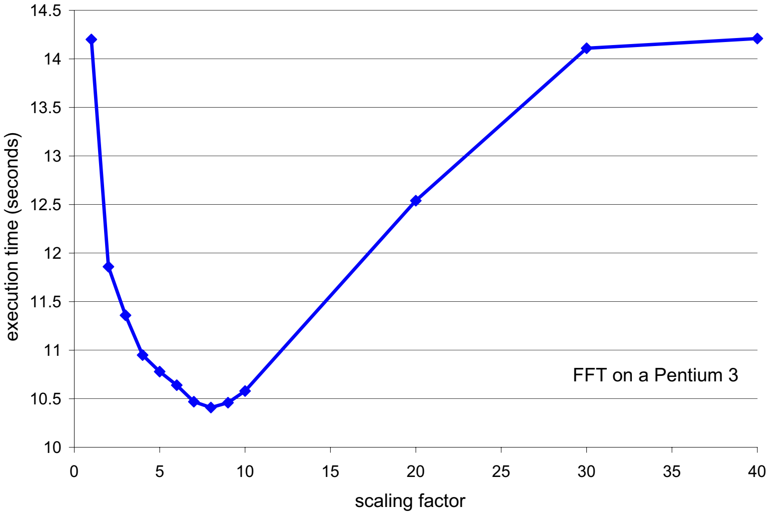


Figure 3-1: Impact of execution scaling on performance.

state. Mathematically, we can substitute into Equation 3.2:

$$\begin{aligned} IMR(S^m) &= \frac{1}{|S^m|}\sum_{i=1}^{|S^m|} IMISS(S^m, i) \\ &= \frac{1}{m \times |S|}\sum_{i=1}^{|S^m|} IMISS(S^m, i). \\ &= \frac{1}{m \times |S|}\sum_{i=1}^{|S|} IMISS(S, i). \end{aligned} \tag{3.3}$$

The last step is possible because *IMISS* is zero for $m-1$ out of m executions of every scaled actor. The result is that the miss rate is inversely proportional to the scaling factor.

In Figure 3-1 we show a representative curve relating the scaling factor to overall performance. The data corresponds to a coarse-grained implementation of a Fast Fourier Transform (FFT) running on a Pentium 3 architecture. The x-axis represents the scaling factors (with increasing values from left to right). The y-axis represents the execution time and is an indirect indicator of the miss rate (the two measures are positively correlated). The execution time improves in accord with the model: the running time is shortened as the scaling factor grows larger. There is however an eventual degradation, and as the sequel will show, it is attributed to the data cache performance.

3.2 Data Cache

The results in Figure 3-1 show that scaling can reduce the running time of a program, but ultimately, it degrades performance. This section provides a basic analytical model that helps in reasoning about the relationship between scaling and the data cache miss rate.

We distinguish between two types of data working sets. The static data working set of an actor represents state, e.g., `weights` in the FIR example (Figure 2-1). The dynamic data working set is the data consumed (poped) from the input channel and generated (pushed) to the output channel by the work function. Both of these working sets impact the data cache behavior of an actor.

Intuitively, the presence of state suggests that it is prudent to maximize that working set's temporal locality. In this case, scaling positively improves the data cache performance. To see that this is true, we can define a data miss rate (DMR) based on a derivation similar to that for the instruction miss rate, replacing C_I with C_D in Equation 3.1, and $I(a)$ with $State(a)$ when calculating the reuse distance. Here, C_D represents the data cache size, and $State(a)$ represents the total size of the static data in the specified actor.

Execution scaling however also increases the I/O requirements of a scaled actor. Let *pop* and *push* denote the declared pop and push rates of an actor, respectively. The scaling of an actor by a factor m therefore increases the pop rate to $m \times pop$ and the push rate to $m \times push$. Combined, we represent the dynamic data working set of an actor a as $IO(a, m) = m \times (pop + push)$. Therefore, we measure the data reuse distance (DRD) of an execution S with scaling factor m as follows:

$$DRD(S^m, i) = \sum_a State(a) + IO(a, m)$$

where the sum is over all distinct actors a occurring in $\mathit{phase}(S^m, i)$. While this simple measure double-counts data that are both produced and consumed within a phase, such duplication could be roughly accounted for by using $\mathit{IO}'(a, m) = \mathit{IO}(a, m)/2$.

We can determine if a specific work function will result in a data cache miss (on its next firing) by evaluating the following step function:

$$\mathit{DMISS}(S^m, i) = \begin{cases} 0 & \text{if } \mathit{DRD}(S^m, i) \leq C_D\text{; hit: no cache refill,} \\ 1 & \text{otherwise; miss: (some) cache refill.} \end{cases} \tag{3.4}$$

Finally, to model the data miss rate (DMR):

$$\mathit{DMR}(S^m) \;=\; \frac{1}{|S^m|} \sum_{i=1}^{|S^m|} \mathit{DMISS}(S^m, i). \tag{3.5}$$

It is evident from Equation 3.5 that scaling can lead to lower data miss rates, as the coefficient $1/|S^m| = 1/(m \times |S|)$ is inversely proportional to m. However, as the scaling factor m grows larger, more of the DMISS values transition from 0 to 1 (they increase monotonically with the I/O rate, which is proportional to m). For sufficiently large m, $\mathit{DMR}(S^m) = 1$. Thus, scaling must be performed in moderation to avoid negatively impacting the data locality.

Note that in order to generalize the data miss rate equation so that it properly accounts for the dynamic working set, we must consider the amount of data reuse within a phase. This is because any actor that fires within *phase(S,i)* might consume some or all of the data generated by $S[i]$. The current model is simplistic, and leads to exaggerated I/O requirements for a phase. We also do not model the effects of cache conflicts, and take an "atomic" view of cache misses (i.e., either the entire working set hits or misses).

Chapter 4

Optimization Algorithm

In this chapter we describe our memory optimizations that are geared toward improving the memory behavior of streaming programs. First, we describe *cache aware fusion* which performs a series of granularity adjustments to the actors in the steady state. The fusion serves to (*i*) reduce the overhead of switching between actors, (*ii*) create coarser grained actors for execution scaling, and (*iii*) enable novel buffer management techniques between fused actors. Second, we describe *execution scaling* which scales a steady state to improve instruction locality, subject to the data working set constraints of the actors in the stream graph. Third, we describe scalar replacement which enables register allocation of intermediate values that are passed between fused filters. Last, we discuss an optimized management strategy for the data in the FIFO channels to support peeking, that reduce the number of memory accesses without introducing substantial computational overhead.

4.1 Cache Optimizations

The cache aware fusion in conjunction with execution scaling represent a *unified cache optimization* that simultaneously considers the instruction and data working sets of actors that make up a stream program.

4.1.1 Cache Aware Fusion

In StreamIt, the granularity of actors is determined by the application developer, according to the most natural representation of an algorithm. When compiling to a cache-based architecture, the presence of a large number of actors exacerbates the transition overhead between work functions. It is the role of the compiler to adjust the granularity of the stream graph to mitigate the execution overhead.

In this section we describe an actor coarsening technique we refer to as *cache aware fusion* (CAF). When two actors are fused, they form a new actor whose work function is equivalent to its constituents. For example, let an actor `A` fire n times, and an actor `B` fire $2n$ times per steady state: $S^n = (\mathtt{A}^n\mathtt{B}^n\mathtt{B}^n)$. Fusing `A` and `B` results in an actor `F` that is equivalent to one firing of `A` and two firings of `B`; `F` fires n times per steady state ($S^n = (\mathtt{F}^n)$). In other terms, the work function for actor `F` inlines the work functions of `A` and `B`.

When two actors are fused, their executions are scaled such that the output rate of one actor matches the input rate of the next. In the example, `A` and `B` represent a producer-consumer pair of filters within a pipeline, with filter `A` pushing two items per firing, and `B` popping one item per firing. The fusion implicitly scales the execution of `B` so that it runs twice for every firing of `A`.

Fusion also reduces the overhead of switching between work functions. In our infrastructure, the steady state is a loop that invokes the work functions via method calls. Thus, every pair of fused actors eliminates a method call (per invocation of the actors). The impact on performance can be significant, but not only because method calls are removed: the fusion of two actors also enables the compiler to optimize across actor boundaries. In particular, for actors that exchange only a few data items, the compiler can allocate the data streams to registers. The data channel between fused actors is subject to special buffer management techniques (e.g. scalar replacement) as described in the Section 4.2.

There are, however, downsides to fusion. First, as more and more actors are fused, the instruction footprint can dramatically increase, possibly leading to poor use of the instruction cache. Second, fusion increases the data footprint when the fused actors maintain state (e.g., coefficient arrays and lookup tables). Our fusion algorithm is cache aware in that it is cognizant of the instruction and data sizes.

The CAF algorithm uses a greedy fusion heuristic to determine which filters should be fused. It continuously fuses actors until the addition of a new actor causes the fused actor to exceed *either* the instruction cache capacity, or a fraction of the data cache capacity. For the former, we estimate the instruction code size using a simple count of the number of operations in the intermediate representation of the work function. For the latter, we allow the state of the new fused actor to occupy up to some fraction of the data cache capacity (e.g. 50%).

The algorithm illustrated in Figure 4-1 leverages the hierarchical nature of the stream graph, starting at the leaf nodes and working upward. For pipeline streams, the algorithm identifies the connection in the pipeline with the highest steady-state I/O rate, i.e., the pair of filters that communicate the largest number of items per steady state. These two filters are fused, if doing so respects the instruction and data cache constraints. To prevent fragmentation of the pipeline, each fused filter is further fused with its upstream and downstream neighbors so long as the constraints are met. The algorithm then repeats this process with the next highest-bandwidth connection in the pipeline, continuing until no more filters can be fused. For splitjoin streams, the CAF algorithm fuses all parallel branches together if the combination satisfies the instruction and data constraints. Partial fusion of a splitjoin is not helpful, as the child streams do not communicate directly with each other; however, complete fusion can enable further fusion in parent pipelines.

```
// Following recursive algorithm can be used to find a set of
// partitions for each pipeline or splitjoin such that all actors
// within a partition can be fused without violating instruction or
// data cache constraint.

// For each partition that is returned for the top level pipeline
// all actors within that partition are fused into a new actor.

// To find partitions for a pipeline:

Calculate the number of partitions required for each child,
if for any child this is $> 1$ then remember those partitions.

For each sequence $(i..j)$ of children where for each child
number of partitions is 1 use function Interval($i$,$j$) to find partitions.

Interval($i$,$j$) = “
   Find maximum bandwidth connection between children (in the
   interval $(i..j)$), let this be a pair $m$ and $m + 1$.

   Estimate instruction and data footprint of fused $m$ and $m + 1$.

   If fused $m$ and $m + 1$ violates any cache constraint, use Interval($i$,$m$)
   and Interval($m + 1$,$j$) to find two sets of partitions.

   If fused $m$ and $m + 1$ do not violate any cache constraint, start with
   $m$ and $m + 1$ fused, try fusing up or down until can not fuse up or down
   without violating a cache constraint. Let this result in a partition $(a..b)$,
   use Interval($i$,$a - 1$) and Interval($b + 1$,$j$) to find remaining partitions.”

// To find partitions for a splitjoin:

Calculate the number of partitions required for each child.

If each child can be fused into a single partition, estimate instruction and
data footprint of a fused splitjoin, if this does not violate any cache
constraint return a single partition.

Otherwise, return the set of partitions required for each child.
```

Figure 4-1: Outline of the cache aware fusion algorithm

```
// Returns a scaling factor for steady state $S$
// - $c$ is the data cache size
// - $\alpha$ is the fraction of $c$ dedicated for I/O
// - $p$ is the desired percentile of all actors to be
//   satisfied by the chosen scaling factor $(0 < p \leq 1)$
calculateScalingFactor($S$, $c$, $\alpha$, $p$) {
  create array $M$ of size $|S|$
  for $i = 1$ to $|S|$ {
    $a = S[i]$
    // calculate effective cache size
    $c' = \alpha \times (c - State(a))$
    // calculate scaling factor for $a$ such
    // that I/O requirements are close to $c'$
    $M[i] = \textbf{round}(c' \; / \; IO(a, 1))$
  }
  sort $M$ into ascending numerical order
  $i = \lfloor \; (1 - p) \times |S| \; \rfloor$
  return $M[i]$
}
```

Figure 4-2: Our heuristic for calculating the scaling factor.

4.1.2 Execution Scaling

After we have applied the cache aware fusion algorithm the next step is to scale the granularity adjusted actors in order to reduce the cache-miss rate. According to our instruction cache model, increasing the number of consecutive firings of the same actor leads to lower instruction cache miss rates. However, scaling increases the data buffers that are maintained between actors. Thus it is prudent that we account for the data working set requirements as we scale a steady state.

Our approach is to scale the entire steady state by a single scaling factor, with the constraint that only a small percentage of the actors are allowed to overflow the data cache. Our two-staged algorithm is outlined in Figure 4-2.

First, the algorithm calculates the largest possible scaling factor for every actor that appears in the steady state. To do this, it calculates the amount of data consumed and produced by each actor firing and divides the available data cache size by this data production rate. In addition, the algorithm can toggle the effective cache size to account for various eviction policies.

Second, it chooses the largest factor that allows a fraction p of the steady state actors to be scaled safely (i.e., the cache is adequate for their I/O requirements). For example, the algorithm might calculate $m_{\mathtt{A}} = 10$, $m_{\mathtt{B}} = 20$, $m_{\mathtt{C}} = 30$, and $m_{\mathtt{D}} = 40$, for four actors in some steady state. That is, scaling actor `A` beyond 10 consecutive iterations will cause its dynamic I/O requirements to exceed the data cache. Therefore, the largest m that allows $p = 90\%$ of the actors to be scaled without violating the cache constraints is 10. Similarly, to allow for the safe scaling of $p = 75\%$ of the actors, the largest factor we can choose is 20.

In our implementation, we use a 90-10 heuristic. In other words, we set $p = 90\%$. We empirically determined this value via a series of experiments using our benchmark suite. See Appendix A for an experimental evaluation of our heuristic.

Note that our algorithm adjusts the effective cache size that is reserved for an actor's dynamic working set (i.e., data accessed via `pop` and `push`). This adjustment allows us to control the fraction of the cache that is used for reading and writing data—and affords some flexibility in targeting various cache organizations. For example, architectures with highly associative and multilevel caches may benefit from scaling up the effective cache size (i.e., $\alpha > 1$), whereas a direct mapped cache that is more prone to conflicts may benefit from scaling down the cache (i.e., $\alpha < 1$). In our implementation, we found $\alpha = 2/3$ to work well on desktop processors Pentium 3 and Itanium 2, and $\alpha = 4/3$ to work well on an embedded processor StrongARM 1110. We note that the optimal choice for the effective cache size is a complex function of the underlying cache organization and possibly the application as well; this is an interesting issue that warrants further investigation.

```
void->void pipeline Program {
  add Source();
  add Printer();
}

void->int filter Source {
  int i;
  init { i = 0; }
  work push 2 { push(i++); push(i++); }
}

void->int filter Printer {
  work pop 3 { print(pop() + pop() + pop()); }
}
```

Figure 4-3: Example StreamIt code

4.2 Scalar Replacement

After two filters have been fused into a single work function using *cache aware fusion*, the buffer that contains the intermediate values can be replaced by a set of scalar variables. Such transformation allows the C compiler to register allocate intermediate values and it also eliminates the need to keep track of the current index within the buffer while adding to or removing items from the buffer. In order for the *scalar replacement* to be possible all instructions that access the buffer must access it with a constant index. As our example will show, we can guarantee this property by performing sufficient loop unrolling.

4.2.1 Scalar Replacement Example

Consider a StreamIt program shown in Figure 4-3. The program consists of two filters (`Source` and `Printer`) that have mis-matched rates (filter `Source` pushes two items and filter `Printer` pops three items). If the two filters are fused the compiler will create a pair of loops to match the production and consumption rates as shown in the Figure 4-4. Note that we can not replace the buffer with scalar variables yet since each instruction that accesses the buffer uses a non-constant subscript. To allow *scalar replacement* we need to fully unroll the loops. Note that the result will have three copies of instructions that correspond to the filter `Source` and two copies of

```
fused_work() {
  int buf[6];
  int pushindex = 0;
  int popindex = 0;

  // execute Source
  for (j = 0; j < 3; j++) {
    buf[pushindex++] = i++;
    buf[pushindex++] = i++;
  }

  // execute Printer
  for (j = 0; j < 2; j++) {
    print(buf[popindex++] + buf[popindex++] + buf[popindex++]);
  }
}
```

Figure 4-4: Generated C code corresponding to the fused filter with no unrolling

```
fused_work() {
  int buf0, buf1, buf2, buf3, buf4, buf5;

  buf0 = i++; buf1 = i++; // execute Source
  buf2 = i++; buf3 = i++; // execute Source
  buf4 = i++; buf5 = i++; // execute Source

  print(buf0 + buf1 + buf2); // execute Printer
  print(buf3 + buf4 + buf5); // execute Printer
}
```

Figure 4-5: Generated C code corresponding to the fused filter with full unrolling and scalar replacement

instructions that correspond to the filter `Printer`. Figure 4-5 shows the C code that has been generated after StreamIt compiler has performed full loop unrolling and *scalar replacement*.

4.2.2 Implications for the Cache Aware Fusion

The goal of fusion is to allow aggressive optimization across actor boundaries. Our cache aware fusion algorithm is modified to only fuse a group of filters if a given unroll limit will allow all intermediate buffers to be scalar replaced. For our StreamIt example in Figure 4-3 the filters `Source` and `Printer` will only be fused if the loop unrolling limit is greater than or equal to 3 (otherwise the loop around the statements corresponding to filter `Source` in the fused work function will not be fully unrolled). For our benchmark suite we use an unrolling limit of 128 to allow as much fusion and scalar replacement as possible. The cache aware fusion algorithm also keeps track of the code size expansion due to loop unrolling, so that the instruction size of the new actor after unrolling does not exceed the instruction cache.

4.2.3 Implications for Unrolling

We need to perform aggressive unrolling to maximize the number of buffers that are replaced by scalars. However, not all loops should be fully unrolled. For example fully unrolling a loop that does not perform any *push* or *pop* operations unnecessarily increases the instruction size of the actor (this may limit our ability to fuse an actor with other actors without exceeding the instruction cache). Therefore loops that do not perform *push* or *pop* operations are unrolled no more than 4 times.

4.3 Optimized Buffering of Live Items

For FIFO channels where the consumer only examines (peeks) the same items that it consumes during each iteration ($peek \leq pop$) a simple buffer of sufficient size can be used. The buffer is first filled up by the producer and subsequently emptied by the consumer. As shown in the previous section such buffers can be replaced with a set of scalar variables if the two filters are fused and sufficient loop unrolling is performed.

For FIFO channels where the consumer examines more items than it consumes ($peek > pop$) the buffer will be primed during the initialization phase to allow the consumer to execute. Subsequently, the buffer is never completely emptied by the consumer. This imposes a difficult decision on our StreamIt compiler of how to best implement a buffer that has to retain a set of live items for consumption during subsequent steady state cycles.

We explore two basic strategies for implementing buffers that must retain live items between steady state executions. The first strategy, termed *modulation*, implements a traditional circular buffer that is indexed via a wrap-around head and tail pointers. The second strategy, termed *copy-shift*, avoids modulo operations by shifting the buffer contents to the start of the buffer after a certain number of executions. Our experimental evaluation demonstrates that, while a naive implementation of copy-shift can be 2× to 3× slower than modulation, optimizations that utilize execution scaling can boost the performance of copy-shift to be significantly faster than modulation (51% speedup on StrongARM, 48% speedup on Pentium 3, and 5% speedup on Itanium 2).

4.3.1 Modulation

The modulation scheme uses a traditional circular-buffer approach. Three variables are introduced: a `BUFFER` to hold all items transferred between the actors, a `push_index` to indicate the buffer location that will be written next, and a `pop_index` to indicate the buffer location that will be read next (i.e., the location corresponding to `peek(0)`). The communication primitives are translated as follows:

```
push(val); ==>  BUFFER[push_index] := val;
                push_index := (push_index + 1) % BUF_SIZE;


pop();     ==>  := BUFFER[pop_index];
                pop_index := (pop_index + 1) % BUF_SIZE;


peek(i)    ==>  := BUFFER[(pop_index + i) % BUF_SIZE]
```

Note that, for performance reasons the StreamIt compiler converts the modulo operations to bitwise-and operations by scaling the buffer to a power of two.

4.3.2 Copy-Shift

A copy-shift implementation allows us to eliminate the bitwise-and operations, by not allowing the head and tail pointers to wrap around the buffer. Instead, the live items are periodically copied to the start of the buffer and the head and tail pointers are decreased. The communication primitives are translated as follows:

```
push(val); ==>  BUFFER[push_index++] := val;


pop();     ==>  := BUFFER[pop_index++];


peek(i)    ==>  := BUFFER[pop_index + i]
```

An unoptimized implementation of copy-shift in our StreamIt compiler copies the live items after each execution of the consumer that has been scaled only to match rates with other fused actors. The cost of copying a substantial amount of data frequently makes the unoptimized copy-shift substantially less efficient than simple modulation as our experimental evaluation will show.

4.3.3 Optimized Copy-Shift

We can reduce the cost of copy-shift by increasing the size of the data buffer. This allows us to reduce the frequency at which the live items are copied over to the beginning of the buffer. We evaluate two transformations of a stream program that allow us to reduce the frequency of copying the live items.

Peek-Scaling Implementation

A simple transformation, that allows us to reduce the number of times the live items are copied, is to replace every filter that peeks (i.e., $peek > pop$) with a filter that executes the original filter N times. N is chosen sufficiently large such that the cost of copying items per execution of the original filter is reduced (since live items will be copied to the beginning of the buffer N times less). After scaling, the new filter has a pop rate equal to $pop_n = N * pop_o$ and a peek rate equal to $peek_n = N * pop_o + (peek_o - pop_o)$, where pop_o and $peek_o$ are the pop and peek rates of the original filter, and pop_n and $peek_n$ are the pop and peek rates of the replaced filter. The compiler choses N such that $(peek_n - pop_n) \leq pop_n/4$ (the original filter is executed N times such that the new filter consumes at least 4× as many items than are copied over to the start of buffer after every iteration of the new filter). As our experimental evaluation will show this transformation allows copy-shift to outperform modulation for our synthetic benchmark. However, this transformation can lead to significant performance reduction for some of our application benchmarks, since the loop that is introduced by the peek-scaling will be unrolled to allow scalar replacement leading to an increase in the instruction footprint. Also the loops enclosing other fused actors will have larger iteration counts to match the new consumption/production rate of the replaced filter; this leads to increased code size due to unrolling. Lastly, the sum of input and output data consumed during a steady state for some actor after the peek-scaling transformation might exceed the size of the data cache leading to bad data cache performance.

Cut-Peek Implementation

Another approach that allows us to decrease the frequency at which live items are copied to the start of the buffer is to modify our cache aware fusion algorithm to never fuse a producer consumer pair if the consumer performs any peeking (i.e., $peek > pop$). This ensures that after we perform execution scaling we can copy the live items only once per execution of the scaled consumer (which might be fused with filters that consume its output). The cut-peek implementation presents a unified optimization framework for reducing cache miss rates and achieving good performance for our copy-shift buffer implementation.

Chapter 5

Experimental Evaluation

In this chapter we evaluate the merits of the proposed memory optimizations and buffer management strategies. We use three different architectures: a 137 MHz StrongARM 1110, a 600 MHz Pentium 3 and a 1.3 GHz Itanium 2. The StrongARM results reflect performance for an embedded target; it has a 16 Kb L1 instruction cache, an 8 Kb L1 data cache, and no L2 cache. The StrongARM also has a separate 512-byte minicache (not targeted by our optimizations). The Pentium 3 and Itanium 2 reflect desktop performance; they have a 16 Kb L1 instruction cache, 16 Kb L1 data cache, and 256 Kb shared L2 cache.

Our benchmark suite (see Table 5.1) consists of 11 StreamIt applications. They are compiled with the StreamIt compiler which applies the optimizations described in this thesis, as well as aggressive loop unrolling (by a factor of 128 for all benchmarks) to facilitate scalar replacement (see Chapter 4). The StreamIt compiler outputs a functionally equivalent C program that is compiled with `gcc` (v3.4, -O3) for the StrongARM and for the Pentium 3, and with `ecc` (v7.0, -O3) for the Itanium 2. Each benchmark is then run five times, and the median user time is recorded.

As the StrongARM does not have a floating point unit, we converted all of our floating point applications (i.e., every application except for `bitonic`) to operate on integers rather than floats. In practice, a detailed precision analysis is needed in converting such applications to fixed-point. However, as the control flow within these applications is very static, we are able to preserve the computation pattern for the

Benchmark	**Description**	**# of Actors**
`bitonic`	bitonic sort of 64 integers	972
`fir`	finite impulse response (128 taps)	132
`fft-fine`	fine grained 64-way FFT	267
`fft-coarse`	coarse grained 64-way FFT	26
`3gpp`	3GPP Radio Access Protocol	105
`beamformer`	beamformer with 64 channels and 1 beam	197
`matmult`	matrix multiplication	48
`fmradio`	FM Radio with 10-way equalizer	49
`filterbank`	filterbank program (8 bands, 32 taps / filter)	53
`filterbank2`	independent filterbank (3 bands, 100 taps / filter)	37
`ofdm`	Orthogonal Frequency Division Multiplexor [26]	16

Table 5.1: Evaluation benchmark suite.

sake of benchmarking by simply replacing every floating point type with an integer type.

We also made an additional modification in compiling to the StrongARM: our execution scaling heuristic scales actors until their output fills 4/3 of the data cache, rather than 2/3 used for the Pentium 3 and the Itanium 2. This modification accounts for the 32-way set-associative L1 data cache in the StrongARM. Due to the high degree of associativity, there is a smaller chance that the actor outputs will repeatedly evict the state variables of the actor, thereby making it worthwhile to further fill the data cache. Note, that, since $4/3 > 1$, we expect the data produced by the actor to overwrite the data consumed without evicting the state. Using 4/3 instead of 2/3 on StrongARM yields up to 30% improvement on some benchmarks.

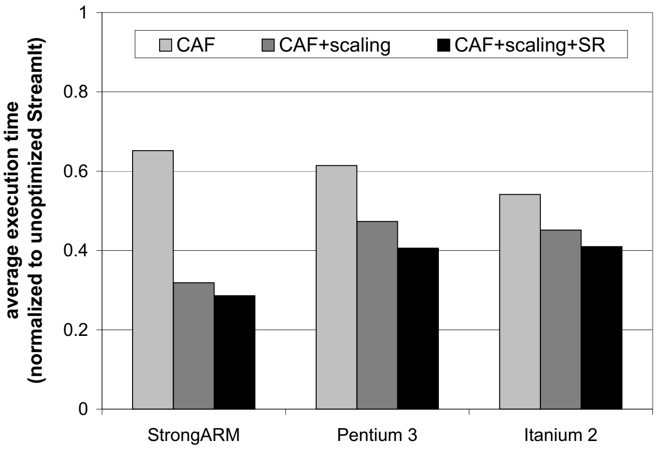


Figure 5-1: Impact on average execution time for our benchmark suite.

5.1 Evaluation of Cache Aware Fusion, Scaling and Scalar Replacement

In this section we evaluate the performance impact of cache aware fusion, execution scaling and scalar replacement on an embedded processor, a superscalar processor and a VLIW processor. Instead of evaluating each optimization individually we first evaluate the performance impact of applying just cache aware fusion (CAF), then the impact of cache aware fusion in combination with execution scaling (CAF+scaling), and lastly the impact of cache aware fusion in combination with execution scaling and scalar replacement within the granularity adjusted actors (CAF+scaling+SR).

Instead of calculating a speedup of an optimization plan using geometric mean of the execution times of individual benchmarks we use average execution time to calculate speedups. We believe that using an average execution time is appropriate instead of using a geometric mean since an average execution time gives an equal weight to the execution time of all eleven benchmarks. Using the geometric mean would actually make all of our speedups over unoptimized StreamIt larger.

Figure 5-1 shows the impact of our optimizations on the average execution time for our benchmark suite on all three architectures. Cache aware fusion alone delivers a speedup of 53% on StrongARM, a speedup of 63% on Pentium 3 and a speedup of 85% on Itanium 2 over unoptimized StreamIt. Cache aware fusion with execution scaling delivers a speedup of 214% on StrongARM, a speedup of 111% on Pentium 3 and a speedup of 122% on Itanium 2 over unoptimized StreamIt. Cache aware fusion with execution scaling and scalar replacement delivers a speedup of 250% on StrongARM, a speedup of 146% on Pentium 3 and a speedup of 144% on Itanium 2 over unoptimized StreamIt.

Figure 5-2, Figure 5-3 and Figure 5-4 show the performance impact of applying optimizations CAF, CAF+scaling and CAF+scaling+SR for individual benchmarks on all three architectures. At the right-hand side of each figure we show the average and geometric mean of the normalized execution time.

In general we observe that adding execution scaling to cache aware fusion improves performance for all benchmarks on all platforms. The only exception is `3gpp` on StrongARM. This is possibly due to items in flight between the granularity-adjusted actors overwriting the state of an executing actor in the data cache. Since StrongARM has no L2 cache then such eviction can be quite expensive. It could also be due to our scaling algorithm allowing input and output to occupy up to 4/3 of the data cache on StrongARM. However, all other benchmarks on StrongARM have better performance when we allow actors to fill 4/3 of the data cache instead of 3/3 or 2/3. Also note that execution scaling has the most impact on StrongARM architecture, this is possibly due to its lack of an L2 cache that makes cache-misses much more expensive than on Pentium or Itanium which both have a substantial L2 cache.

We also observe that adding scalar replacement to cache aware fusion and execution scaling improves performance for almost all benchmarks on all platforms. The only exceptions are `fft-fine` and `filterbank2` on a StrongARM which experience a modest 15% and 8% slowdown; also matrix multiply experiences a negligible 1% slowdown on a Pentium 3.

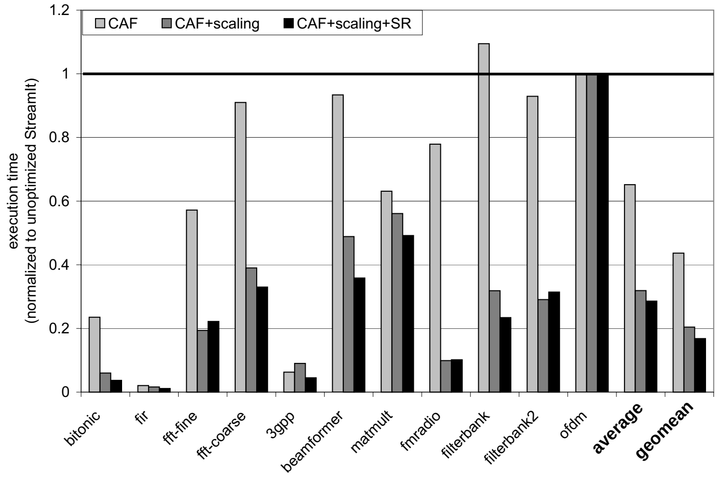


Figure 5-2: Performance results for StrongARM 1110

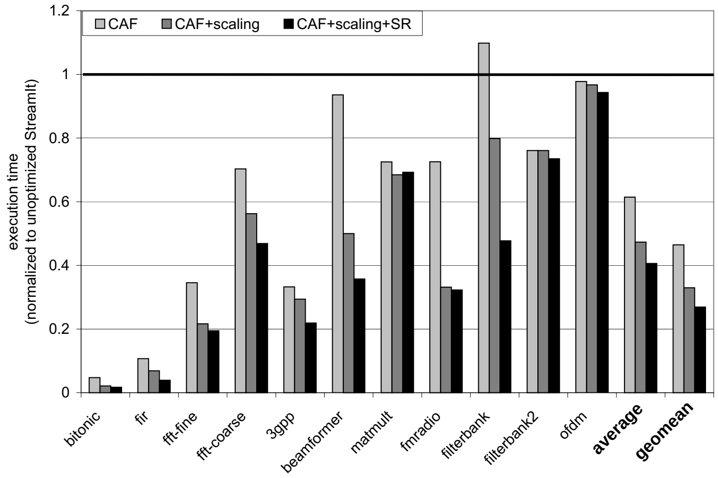


Figure 5-3: Performance results for Pentium 3

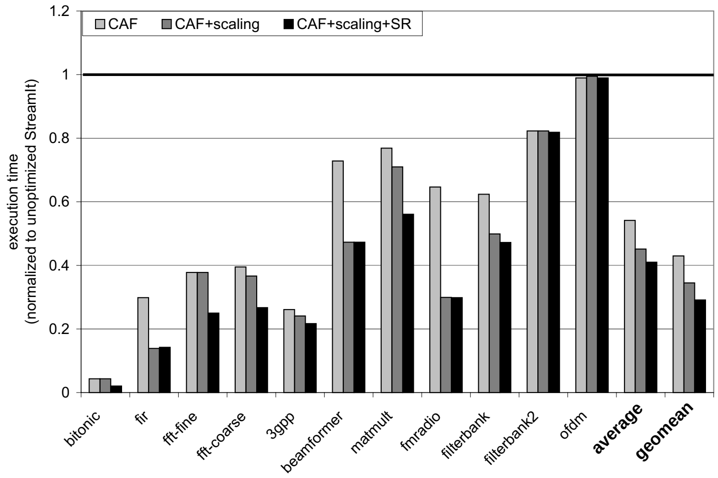


Figure 5-4: Performance results for Itanium 2

5.2 Evaluation of Copy-Shift and Modulation

To compare the efficiency of different buffer management techniques we use a simple synthetic StreamIt benchmark shown in Figure 5-5. Using a specialized synthetic benchmark allows us to highlight the performance of specific buffer management implementation techniques. We compile the benchmark with filter `FIR` having a peek rate set equal to 0, 8, 16, 32 ...128 (by varying the `PEEK` variable). Varying the peek rate of the `FIR` filter allows us to see how the strategies perform as we vary the number of live items that must be retained in the buffer. We ran the benchmark on the StrongARM 1110, Pentium 3 and Itanium 2. The results are shown in Figure 5-6, Figure 5-7 and Figure 5-8. The `copy-shift` represents an unoptimized implementation of copy-shift, where the live items are copied to the beginning of the buffer after every execution of the filter FIR. The `modulation` represents an implementation where the buffer is implemented as a wrap-around buffer. The `copy-shift+scaling` represents peek-scaling where the `FIR` filter is replaced with a filter that executes the original `FIR` filter N times. The N is chosen such that the new filter consumes at least 4× as many items than are copied over to the start of buffer after every iteration of the scaled filter.

As expected, modulation outperforms unoptimized copy-shift, because modulation does not require the items in the buffer to be copied to the start of the buffer. The somewhat surprising result is that optimized copy-shift (where live items are copied to the start of the buffer infrequently) offers a substantial speedup over modulation. The optimized copy-shift in comparison to modulation delivers a 51% speedup on StrongARM, a 48% speedup on Pentium 3, and a 5% speedup on Itanium 2 for a peek rate of 128. The modest speedup of using optimized copy-shift versus modulation on Itanium 2 can be explained by the VLIW nature of the architecture, where the bitwise-and operations can be scheduled by the C compiler in parallel with other instructions, thus reducing the cost of bitwise-and operation relative to its cost on StrongARM and Pentium 3.

```
void->void pipeline BufferTest {
  add Source();
  add FIR();
}

void->float filter Source {
  work push 1 {
    push( ... );
  }
}

float->void filter FIR {
  int PEEK = 4;
  work pop 1 peek PEEK {
    float result = 0;
    for (int i = 1; i < PEEK; i++) {
      result += i * peek(i);
    }
    pop();
    print(result);
  }
}
```

Figure 5-5: Original StreamIt code for the buffer test.

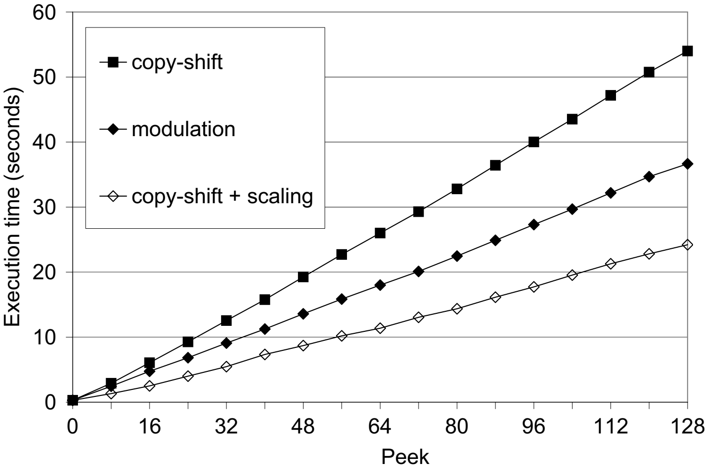


Figure 5-6: Performance of buffer management strategies on a StrongARM 1110

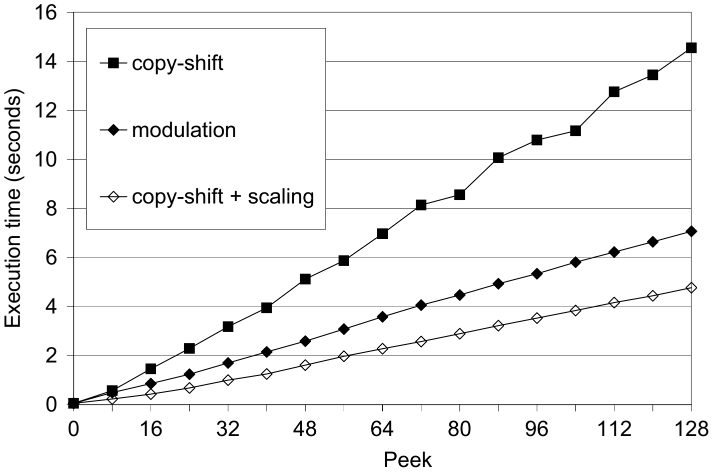


Figure 5-7: Performance of buffer management strategies on a Pentium 3

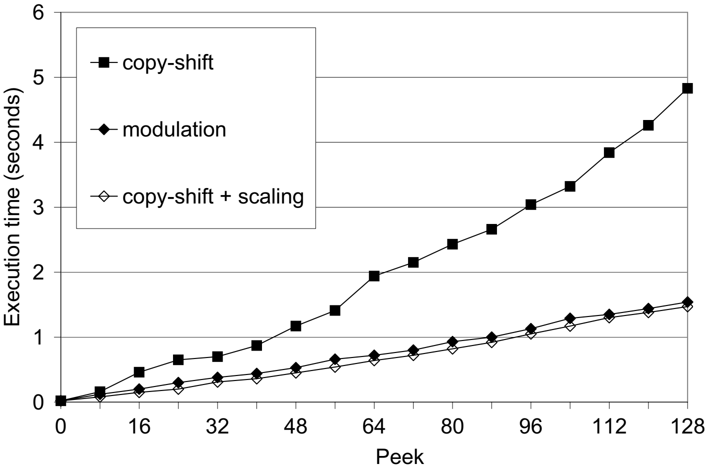


Figure 5-8: Performance of buffer management strategies on an Itanium 2

Table 5.2: The best performing buffer management strategies for each benchmark-architecture pair [along with speedup over CAF+scaling+SR]

Benchmark	**StrongARM 1110**	**Pentium 3**	**Itanium 2**
`fmradio`	peek-scaling 5%	peek-scaling 3%	peek-scaling, cut-peek 0%
`filterbank`	default	default	default
`filterbank2`	cut-peek 45%	cut-peek 38%	cut-peek 36%
`ofdm`	peek-scaling 7%	cut-peek 3%	cut-peek 0%

5.3 Evaluation of Peek-Scaling and Cut-Peek

The previous section suggests that the best implementation for a buffer that has to retain live items is a copy-shift that copies the live items to the start of buffer infrequently. However, the buffer management strategy for the performance numbers we presented in previous sections is unoptimized copy-shift. In this section we evaluate two transformations of a stream program that allow us to use optimized copy-shift by applying a simple transformation to the stream program. The two alternatives are: peek-scaling and cut-peek (see Chapter 4 for details). From eleven benchmarks in our benchmark suite, only four benchmarks have filters that peek (i.e., $peek > pop$). They are `fmradio`, `filterbank`, `filterbank2` and `ofdm`.

Table 5.2 shows the best buffer implementation for each benchmark and architecture pair along with a speedup over CAF+scaling+SR with unoptimized copy-shift. Although for some benchmarks on some architectures the peek-scaling is the best implementation there are certain risks associated with the peek-scaling transformation. Replacing a filter with a scaled version causes many loops that are placed around filters by the fusion (in order to match rates) to have larger iteration counts. Due to our aggressive unrolling (to allow scalar replacement) total code size after peek-scaling can be substantially larger. Also peek-scaling is cache unaware in that it could overscale some actor such that its total input and output processed during a steady state greatly exceed the data cache size, thus causing a significant slowdown.

Figure 5-9, Figure 5-10 and Figure 5-11 show the performance impact of adding peek-scaling or cut-peek to cache aware fusion, execution scaling and scalar replacement. Note the significant slowdowns for some benchmarks due to applying peek-scaling. The experimental evaluation suggests that cut-peek is better than peek-scaling as an optimized buffer implementation. While cut-peek may result in up to a 17% slowdown (for `filterbank` on Pentium 3) it does deliver significant speedups for benchmarks that have filters with large *peek* - *pop* rate difference (i.e., many live items need to be retained in the FIFO buffer). The best speedups that cut-peek delivers over unoptimized copy-shift are for the `filterbank2` benchmark which has filters where *peek* - *pop* = 99 (a 45% speedup on StrongARM, a 38% speedup on Pentium 3 and 36% speedup on Itanium 2).

A combination of cache aware fusion, execution scaling, scalar replacement and cut-peek buffer management yields a 257% speedup on the StrongARM, a 154% speedup on the Pentium 3, and a 152% speedup on Itanium 2 compared to unoptimized StreamIt.

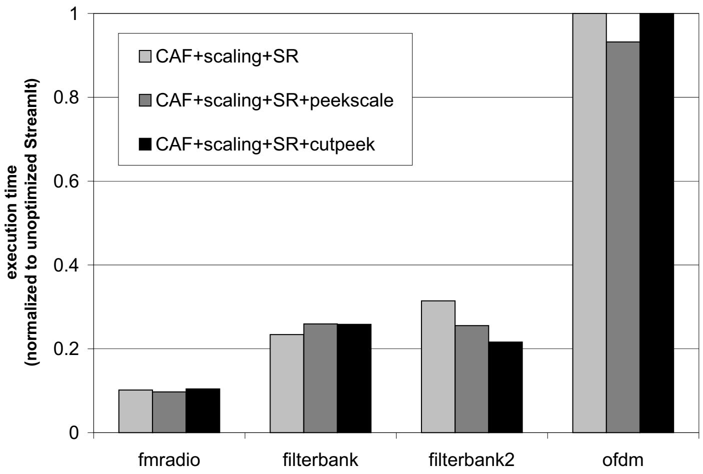


Figure 5-9: Performance of peek-scaling and cut-peek on a StrongARM 1110

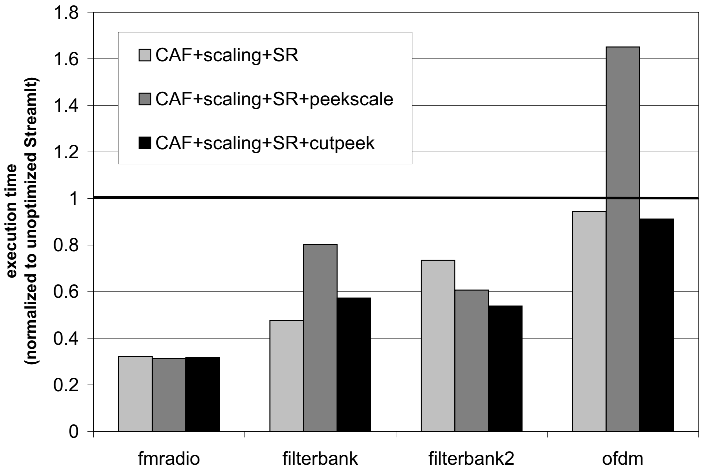


Figure 5-10: Performance of peek-scaling and cut-peek on a Pentium 3

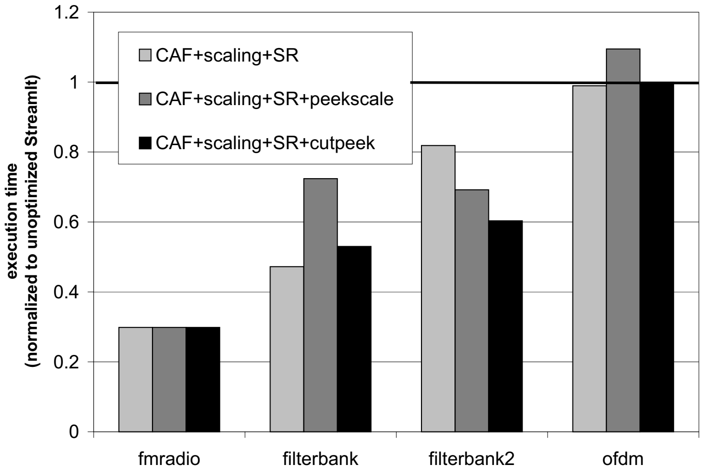


Figure 5-11: Performance of peek-scaling and cut-peek on an Itanium 2

5.4 Comparison to Cache Unaware Full Fusion

An alternative to the optimizations presented in this thesis that allows elimination of method calls and optimizations across actor boundaries is to fuse all actors into a single actor. After combining all actors we can apply scalar replacement to allow intermediate values to be register allocated. However, full fusion is unaware to instruction and data locality, since if we perform aggressive unrolling then there is almost no code reuse, and data locality is enhanced only by executing some actors multiple times to match data production and consumption rates.

On the StrongARM 1110, our cache optimizations offer a 162% speedup over full fusion with scalar replacement (108% speedup if we use geometric mean instead of average, see Figure 5-12). Cache optimizations perform better that full fusion with scalar replacement for all benchmarks except for `3gpp`, where they yield a 45% slowdown. This slowdown is due to conservative code size estimation: the compiler predicts that the fused version of `3gpp` will not fit into the instruction cache, thereby preventing fusion. However, due to optimizations by `gcc`, the final code size is smaller than expected and does fit within the cache. While such inaccuracies could be improved by adding feedback between the output of `gcc` and our code estimation, each fusion possibility would need to be evaluated separately as the fusion boundary affects the impact of low-level optimizations (and thus the final code size).

The speedups offered by cache optimizations over a full fusion strategy are more modest for the desktop processors: 34% speedup on Pentium 3 (17% speedup if we use geometric mean, see Figure 5-13) and essentially zero speedup (6% by the arithmetic mean, -8% by the geometric mean) on Itanium 2 (Figure 5-14). Out of the 11 benchmarks, our cache optimizations perform as well or better than full fusion for 7 benchmarks on the Pentium 3 and 5 benchmarks on the Itanium 2.

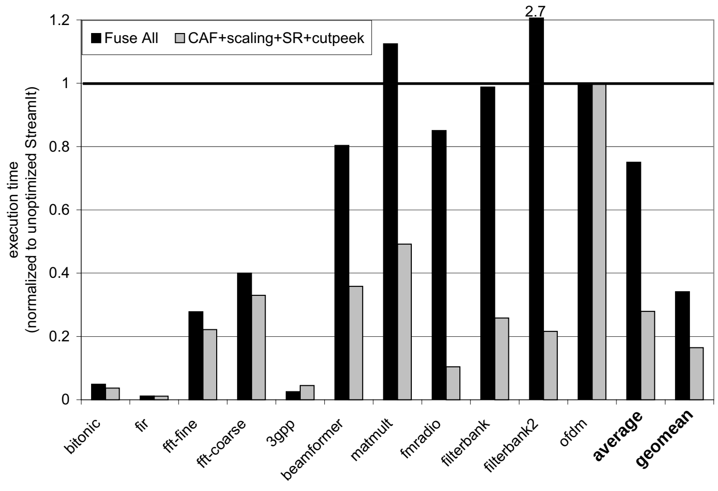


Figure 5-12: Comparison to full fusion on a StrongARM 1110

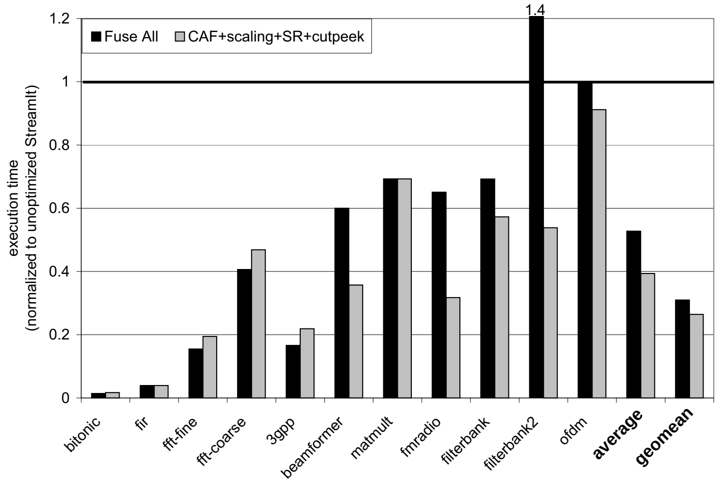


Figure 5-13: Comparison to full fusion on a Pentium 3

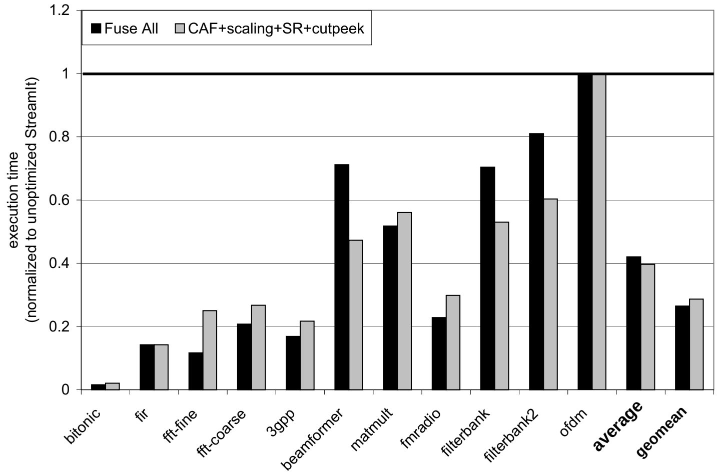


Figure 5-14: Comparison to full fusion on an Itanium 2

5.5 Evaluation of Modified Cache Aware Fusion for Pentium 3 and Itanium 2

Full fusion with scalar replacement outperforms cache optimized executables for `bitonic`, `fft-fine`, `fft-coarse` and `3gpp` on both the Pentium 3 and the Itanium 2 processors, and for `matmult` and `fmradio` benchmarks on the Itanium 2 (see Figure 5-13 and Figure 5-14). Detailed investigation using a hardware performance analyzer (VTune) on the Pentium 3 revealed that as our cache model would predict a fully fused executable for `bitonic`, `fft-coarse` and `fft-fine` has 5× - 10× larger number of cycles during which instruction fetch unit has stalled (due to an instruction cache miss). However, the analyzer also revealed that fully fused executable issues up to 50% less total data memory references. Such a reduction in the number of data memory references must be due to the optimizations enabled in the C compiler by fusion and scalar replacement.

This observation suggests that on the Pentium 3 and the Itanium 2, which have an L2 cache to fall back in cases of instruction fetch miss, it may be beneficial to increase the instruction size limit for actors produced by our cache aware fusion to allow more intermediate value buffers to be scalar replaced. In general, if the overall speedup from the reduction in the number of data memory references is larger than the slowdown due to an increase in the number of instruction fetch misses, then it is beneficial to create actors that do not fit into L1 instruction cache.

Figure 5-15 and Figure 5-16 show the performance of our cache optimizations (CAF+scaling+SR+cutpeek) on a Pentium 3 and an Itanium 2 when the instruction limit for our cache aware fusion algorithm is set to 200Kb (80% of the L2 cache). The only benchmark that is negatively impacted by this change is `ofdm`, where two large actors are fused despite a very low communication to computation ratio, thereby lessening the impact of eliminated memory accesses, while nonetheless worsening the instruction locality and increasing the total instruction size from 47 Kb to 159 Kb (due to unrolling).

The negative impact of our modified cache aware fusion algorithm on `ofdm` sug-

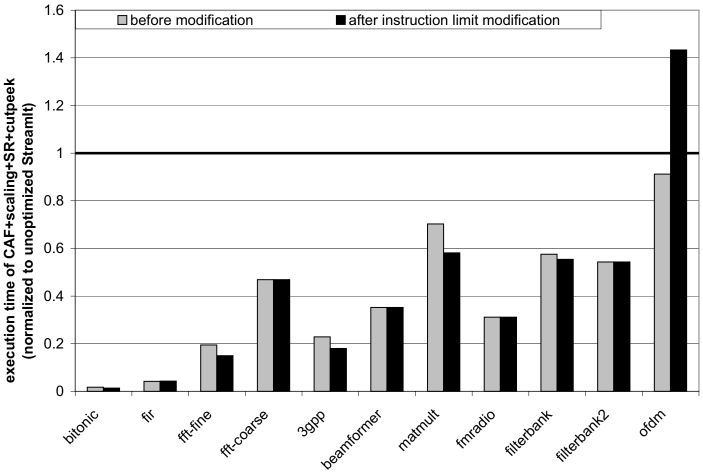


Figure 5-15: Performance of cache optimizations after instruction limit modification on a Pentium 3

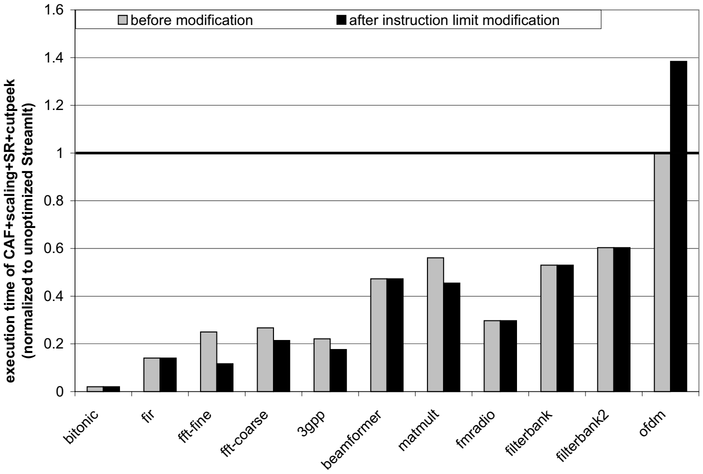


Figure 5-16: Performance of cache optimizations after instruction limit modification on an Itanium 2

gests that in order to avoid the negative performance impact, we need to develop a detailed cost model for evaluating the tradeoff between register allocation due to scalar replacement, and the negative impact of increased code size due to the excessive unrolling that is necessary to enable scalar replacement.

Chapter 6

Related Work

There is a large body of literature on scheduling synchronous dataflow (SDF) graphs to optimize various metrics [4, 5]. The work most closely related to ours is a recent study by Kohli [16] on cache aware scheduling of SDF graphs, implemented as part of the Ptolemy framework for simulating heterogeneous embedded systems [19]. Kohli develops a Cache Aware Scheduling (CAS) heuristic for an embedded target with a software-managed scratchpad instruction cache. His algorithm greedily decides how many times to execute a given actor based on estimates of the data cache and instruction cache penalties associated with switching to the next actor. In contrast, our algorithm considers the buffering requirements of all filters in a given container and increases the multiplicity so long as 90% of buffers are contained within the data cache. Kohli does not consider buffer management strategies, and the evaluation is limited to one 6-filter pipeline and an assortment of random SDF graphs. An empirical comparison of our heuristics on a common architectural target would be an interesting direction for future work.

It is recognized that there is a tradeoff between code size and buffer size when determining an SDF schedule. Most techniques to date have focused on "single appearance schedules" in which each filter appears at only one position in the loop nest denoting the schedule. Such schedules guarantee minimal code size and facilitate the inlining of filters. There are a number of approaches to minimizing the buffer requirements for single-appearance schedules (see [4] for a review). While it has been shown that obtaining the minimal memory requirements for general graphs is NP-complete [3], there are two complimentary heuristics, APGAN (Pairwise Grouping of Adjacent Nodes) and RPMC (Recursive Partitioning by Minimum Cuts), that have been shown to be effective when applied together [3]. Buffer merging[21, 22] represents another technique for decreasing buffer sizes, which could be integrated with our approach in the future.

Govindarajan et al. develop a linear programming framework for determining the "rate-optimal schedule" with the minimal memory requirement [11]. A rate-optimal schedule is one that takes advantage of parallel resources to execute the graph with the maximal throughput. However, the technique is specific to rate-optimal schedules and can result in a code size explosion, as the same node is potentially executed in many different contexts.

The work described above is related to ours in that minimizing buffer requirements can also improve caching behavior. However, our goal is different in that we aim to improve spatial and temporal locality instead of simply decreasing the size of the live data set. In fact, our scaling transformation actually *increases* the size of the data buffers, leading to higher performance across our benchmark suite. Our transformations also take into account the size of the instruction and data caches to select an appropriate scaling and partitioning for the stream graph.

Proebsting and Watterson [23] give a fusion algorithm that interleaves the control flow graphs of adjacent filters. However, their algorithm only supports synchronous `get` and `put` operations; StreamIt's `peek` operation necessitates buffer management between filters.

There are a large number of stream programming languages; see [25] for a review. The Brook language [6] extends C to include data-parallel kernels and multi-dimensional streams that can be manipulated via predefined operators. Synchronous languages such as Esterel [2] and LUSTRE [12] also target the embedded domain, but they are more control-oriented than StreamIt and are less amenable to compile-time optimizations. Benveniste et al. [1] also provides an overview of dataflow synchronous languages. Sisal (Stream and Iteration in a Single Assignment Language) is a high-performance, implicitly parallel functional language [13]. We are not aware of any cache aware optimizations in these stream languages.

There is a large body of work covering cache miss equations, and an equally large body of work concerned with analytical models for reasoning about data reuse distances and cache behavior. The model introduced in this thesis is loosely based on the notion of stack reuse distances [20]. Our model is especially tailored to streaming computations, and unique in leveraging the concept of a steady state execution.

There is some work related to scalar replacement. See [7] for a set of transformations that allow traditional coloring-based register allocator to register allocate individual array elements. Another article [8] presents a fully automated set of transformations that improve memory usage for loops by balancing memory operations and floating-point operations. See [9] for an experimental evaluation of the effectiveness of scalar replacement on scientific benchmarks. While the above papers are concerned with scalar replacement for languages like Fortran and C, this thesis highlights the importance of scalar replacement in a stream compiler, which generally has much more information due to a large fraction of loops with fixed iteration count and lack of aliasing in the StreamIt programming language. Also in a stream program much of the data is communicated using explicit FIFO channels; it is therefore important that as many buffers as possible can be replaced by scalars to enable register allocation.

Chapter 7

Conclusion

This thesis presents a set of simple yet effective cache optimizations that are aimed at improving runtime performance and energy requirements for executing stream programs on commodity processors. We exploit the property of stream programs that allows actors in a stream graph to be freely combined and reordered. This allows the stream compiler to automatically perform the kind of transformations that are often tediously carried out manually for today's programs. Those transformations are otherwise too complex to perform automatically in hardware or in the most aggressive of C compilers.

The transformations presented in this thesis are: *(i)* cache aware fusion, which combines adjacent actors into a single function thus allowing the C compiler to optimize across actor boundaries and reducing the method call overhead. *(ii)* execution scaling, which judiciously repeats actor executions to improve instruction and actor state locality, *(iii)* scalar replacement, which converts certain data buffers into a sequence of scalar variables that can be register allocated, and *(iv)* optimized buffer management, which reduces the overall number of memory accesses issued by the program. The above transformations were implemented as part of StreamIt, a language and compiler infrastructure for stream programming [27].

Finally an experimental evaluation of a fully automated implementation of the cache and memory optimizations shows significant performance improvements over unoptimized StreamIt and cache oblivious full-fusion on an embedded processor StrongARM 1110. The performance gains over cache oblivious full-fusion are more modest for the desktop processors Pentium 3 and Itanium 2.

7.1 Future Work

The 90-10 heuristic which is used for execution scaling might be improved by considering work estimates of stream actors instead of treating all actors as equal. Also on some architectures it might be worth to change 90-10 ratio to 75-25 or some other ratio to allow more scaling at the expense of overscaling larger fraction of actors.

An alternative optimized FIFO buffer implementation would be to increase the size of the buffer and insert if statements that check if the head pointer is close to the end of the data buffer; if so the live items would be copied to the start of the buffer. It might be beneficial to place the if statements outside of the execution scaled actors so that they are invoked infrequently.

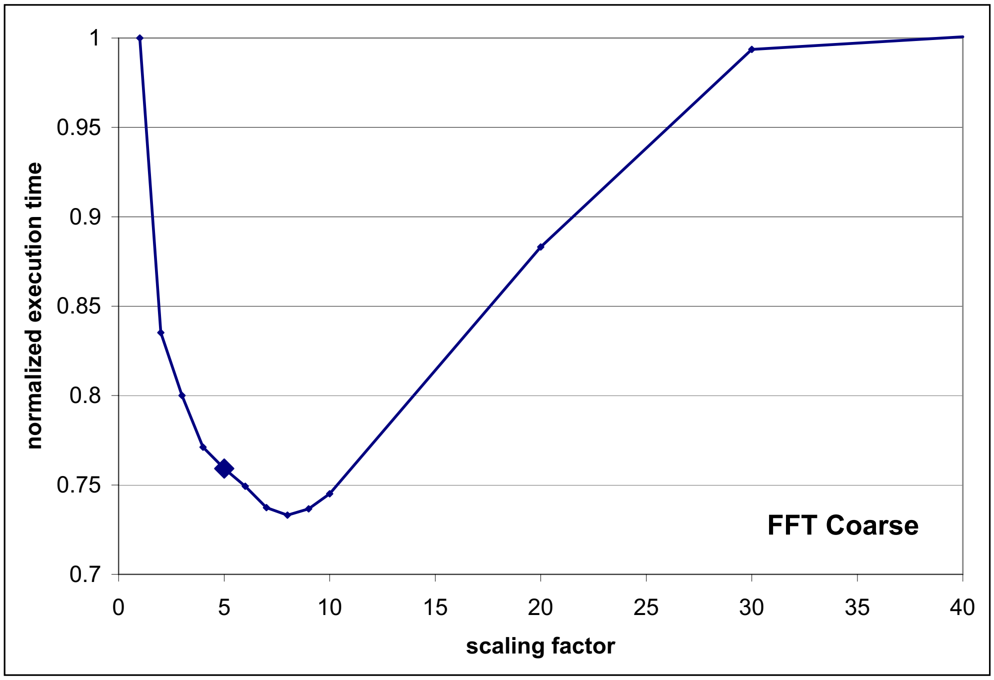
Our experimental evaluation showed that sometimes it is beneficial to create actors with an instruction footprint that exceeds the instruction cache size so that more aggressive optimizations across actor boundaries can be performed. We would need to develop an accurate cost model to allow full automation of such decisions in the compiler.

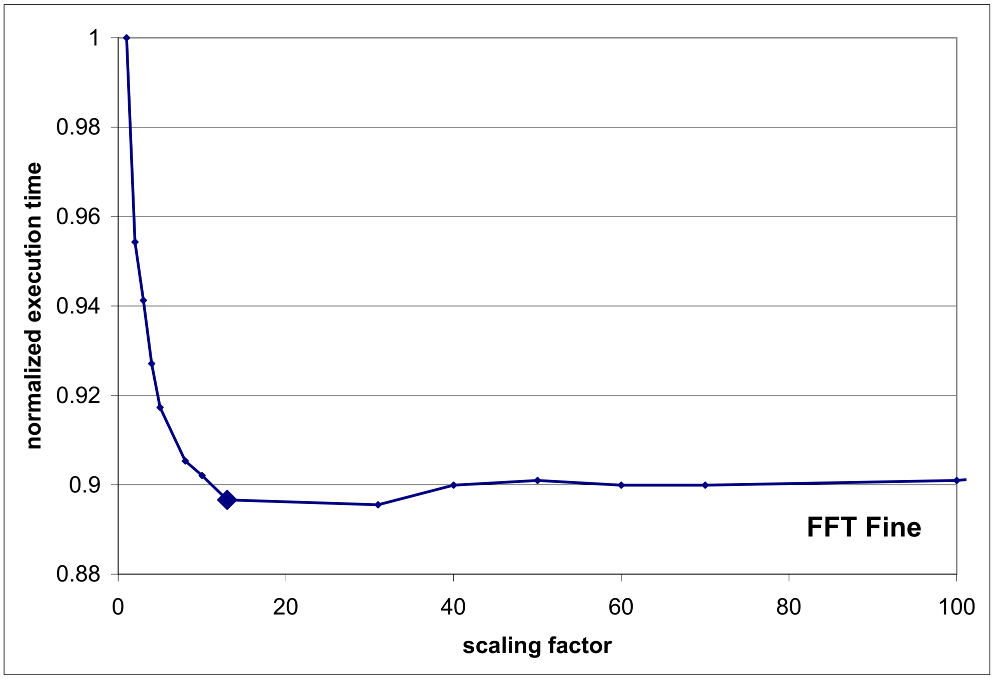
The author of this thesis has also developed a StreamIt backend for a cluster of workstations during his time with the StreamIt group. It would be interesting to see if the cache optimizations presented in this thesis could be used to improve runtime of stream programs on a cluster (by applying cache optimizations to actors that are running on a given cluster node).

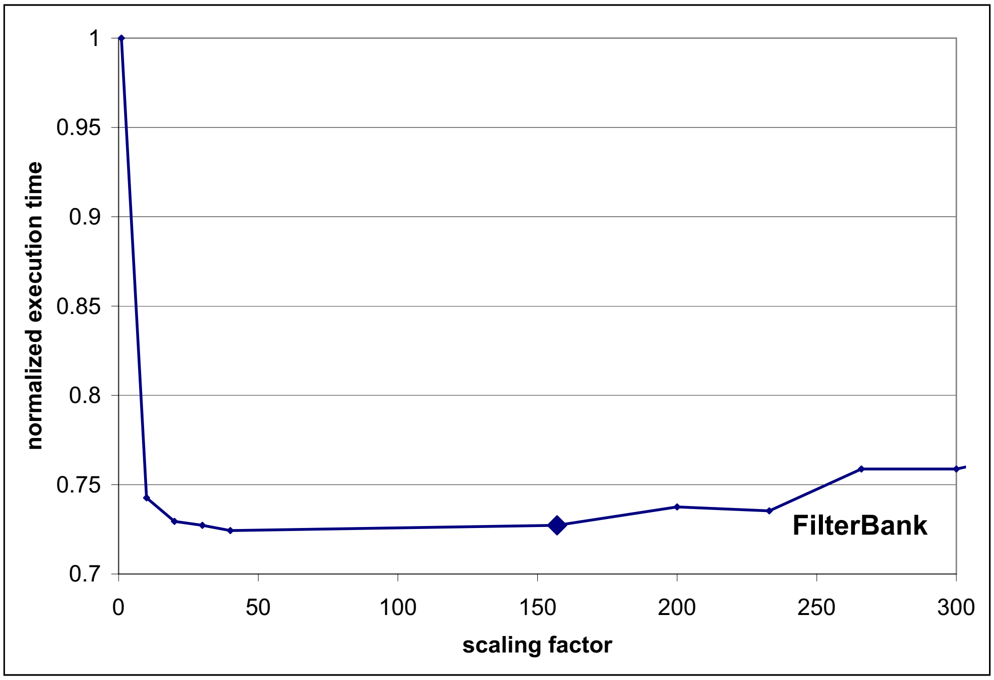
Appendix A

Experimental Evaluation of Execution Scaling Heuristic

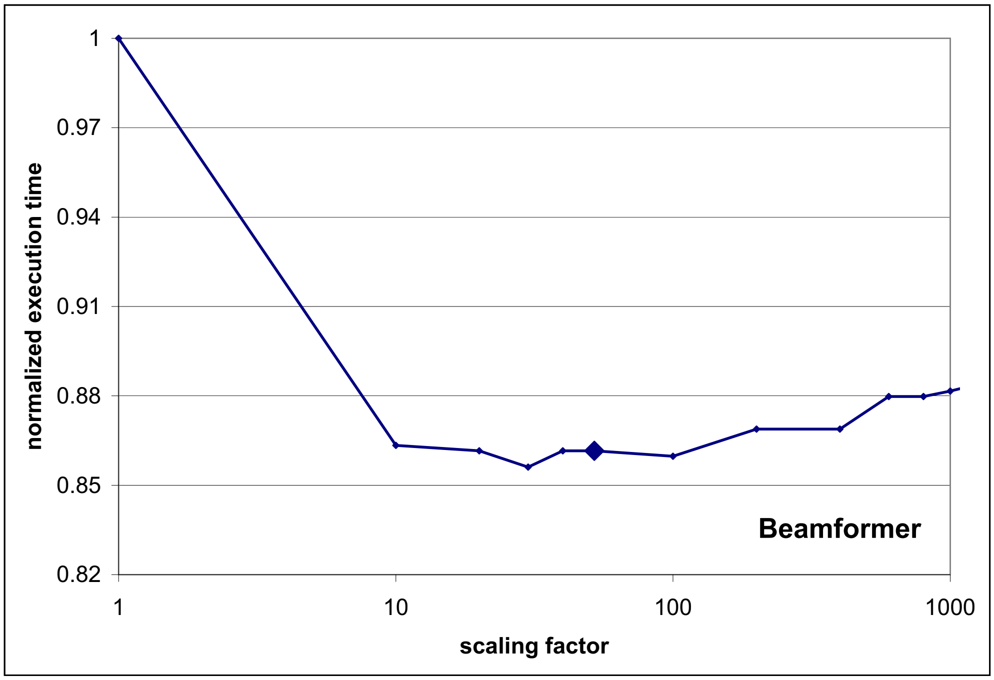
In this chapter we evaluate our execution scaling heuristic (see Section 4.1.2). Following nine graphs show the normalized execution time of scaling cache aware partitions that have been produced using cache aware fusion (without allowing the instruction footprint of a partition to exceed the size of an L1 instruction cache). The experiments are performed on a Pentium 3 processor. The large diamond represents the scaling factor that has been chosen by our 90-10 heuristic.


normalized execution time
1
0.95
0.9
0.85
0.8
0.75
0.7
0
5
10
15
20
25
30
35
40
scaling factor
FFT Coarse

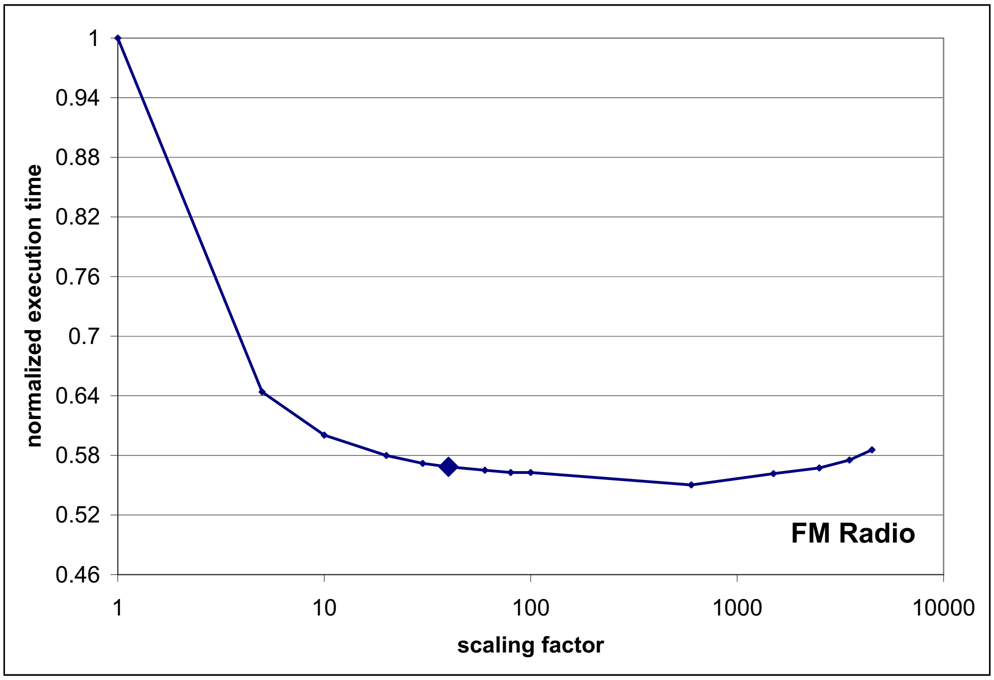

normalized execution time
1
0.98
0.96
0.94
0.92
0.9
0.88
0
20
40
60
80
100
scaling factor
FFT Fine

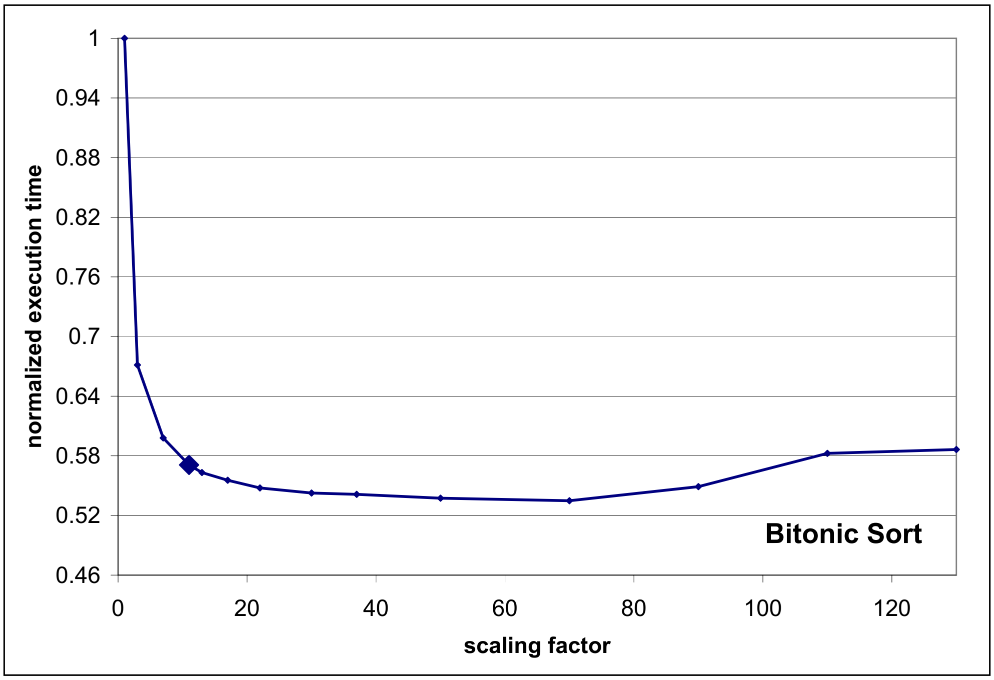


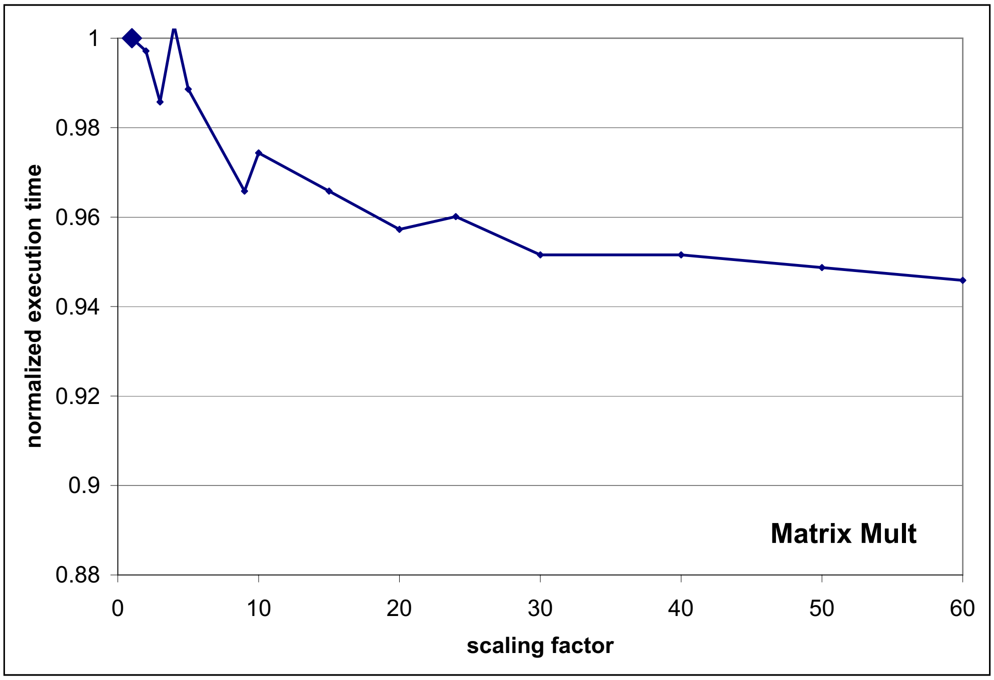
1
0.95
0.9
0.85
0.8
0.75
0.7
normalized execution time
0
50
100
150
200
250
300
scaling factor
FilterBank

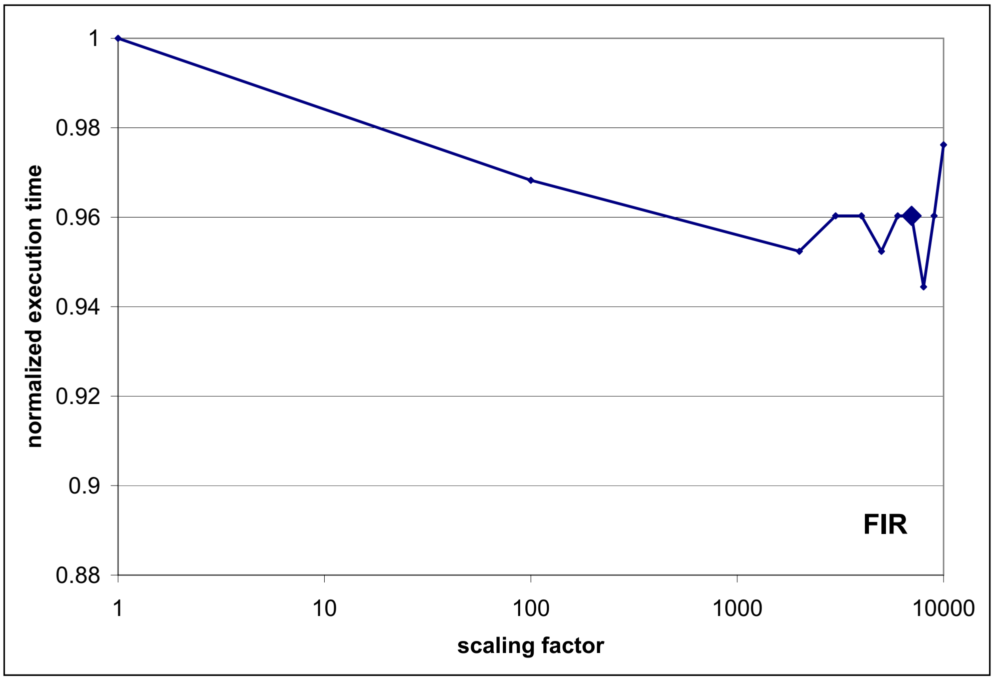


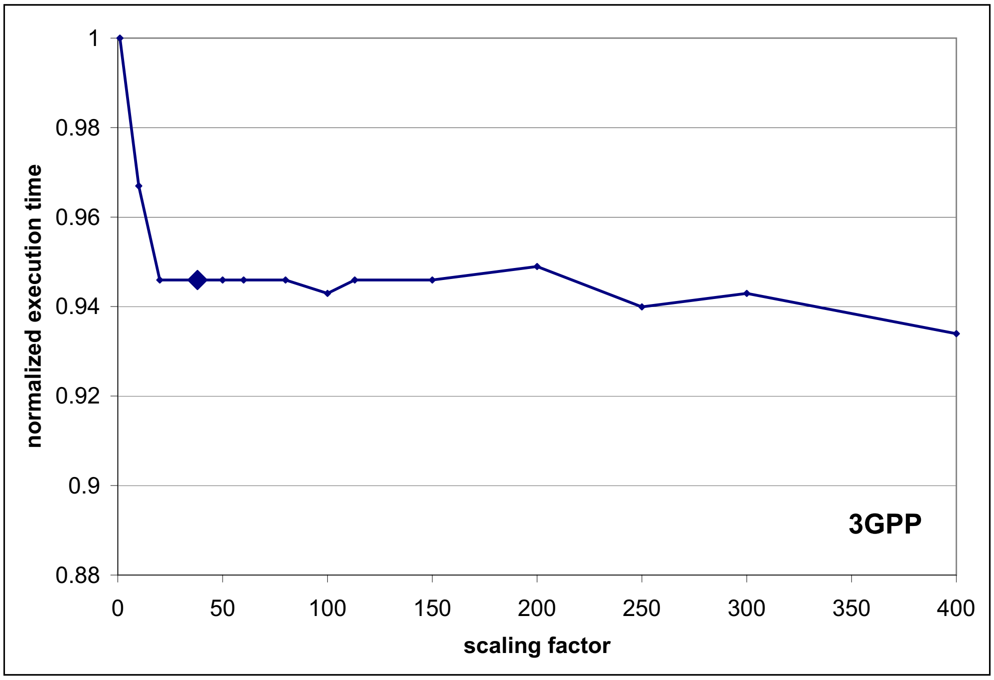
1
0.97
0.94
0.91
0.88
0.85
0.82
normalized execution time
1
10
100
1000
scaling factor
Beamformer


FM Radio
normalized execution time
1
0.94
0.88
0.82
0.76
0.7
0.64
0.58
0.52
0.46
1
10
100
1000
10000
scaling factor


Bitonic Sort
normalized execution time
1
0.94
0.88
0.82
0.76
0.7
0.64
0.58
0.52
0.46
0
20
40
60
80
100
120
scaling factor


normalized execution time
1
0.98
0.96
0.94
0.92
0.9
0.88
0
10
20
30
40
50
60
scaling factor
Matrix Mult


normalized execution time
1
0.98
0.96
0.94
0.92
0.9
0.88
1
10
100
1000
10000
scaling factor
FIR


1
0.98
0.96
0.94
0.92
0.9
0.88
normalized execution time
0
50
100
150
200
250
300
350
400
scaling factor
3GPP

Bibliography

[1] Albert Benveniste, Paul Caspi, Paul Le Guernic, and Nicolas Halbwachs. Data-Flow Synchronous Languages. In *REX School/Symposium*, pages 1–45, 1993.

[2] Gerard Berry and Georges Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Prog.*, 19(2), 1992.

[3] Chuvra S. Bhattacharyya, Praveen K. Murthy, and Edward A. Lee. APGAN and RPMC: Complementary Heuristics for Translating DSP Block Diagrams into Efficient Software Implementations. *Journal of Design Automation for Embedded Systems.*, pages 33–60, January 1997.

[4] S. Bhattacharyya, P. Murthy, and E. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2), June 1999.

[5] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs.* Kluwer Academic Publishers, 1996.

[6] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. In *SIGGRAPH*, 2004.

[7] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. In *PLDI*, pages 53–65, 1990.

[8] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Trans. Program. Lang. Syst.*, 16(6):1768–1810, 1994.

[9] Steve Carr and Philip Sweany. An experimental evaluation of scalar replacement on scientific benchmarks. *Softw. Pract. Exper.*, 33(15):1419–1445, 2003.

[10] Michael Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, 2002.

[11] R. Govindarajan, G.R. Gao, and P. Desai. Minimizing memory requirements in rate-optimal schedules. In *Proceedings of the 1994 Int. conference on Application Specific Array Processors*, pages 75–86, August 1994.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proc. of the IEEE*, 79(9), 1991.

[13] J. Gaudiot and W. Bohm and T. DeBoni and J. Feo and P. Mille. The Sisal Model of Functional Programming and its Implementation. In *Proc. of the Second Aizu International Symposium on Parallel Algorithms/Architectures Synthesis*, 1997.

[14] Michael A. Karczmarek. Constrained and phased scheduling of synchronous data flow graphs for the streamit language. Master's thesis, MIT CSAIL, October 2002.

[15] Michal Karczmarek, William Thies, and Saman Amarasinghe. Phased scheduling of stream programs. In *LCTES*, 2003.

[16] Sanjeev Kohli. Cache aware scheduling of synchronous dataflow programs. Master's Report Technical Memorandum UCB/URL M04/03, UC Berkeley, 2004.

[17] Andrew A. Lamb. Linear analysis and optimization of stream programs. Master's thesis, MIT CSAIL, May 2003.

[18] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, January 1987.

[19] Edward A. Lee. Overview of the Ptolemy Project. Technical report, Tech Memo UCB/ERL M03/25, UC Berkeley, 2003.

[20] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 1970.

[21] P. K. Murthy and S. S. Bhattacharyya. A Buffer Merging Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications. In *International Symposium on System Synthesis*, 1999.

[22] P. K. Murthy and S. S. Bhattacharyya. Buffer Merging — A Powerful Technique for Reducing Memory Requirements of Synchronous Dataflow Specifications. Technical report, Inst. for Adv. Computer Studies, UMD College Park, 2000.

[23] Todd A. Proebsting and Scott A. Watterson. Filter Fusion. In *POPL*, 1996.

[24] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *LCTES*, 2005.

[25] Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7), 1997.

[26] D. Tennenhouse and V. Bose. The SpectrumWare Approach to Wireless Signal Processing. Wireless Networks, 1999.

[27] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proc. of the Int. Conf. on Compiler Construction (CC)*, 2002.

[28] William Thies, Jasper Lin, and Saman Amarasinghe. Partitioning a structured stream graph using dynamic programming. In *5th Workshop on Media and Streaming Processors*, December 2003.