Compiler-Controlled Memory

Keith D. Cooper and Timothy J. Harvey[†]

Computer Science Department
Rice University
6100 Main Street, MS 132
Houston, Texas 77005

Abstract

Optimizations aimed at reducing the impact of memory operations on execution speed have long concentrated on improving cache performance. These efforts achieve a reasonable level of success. The primary limit on the compiler's ability to improve memory behavior is its imperfect knowledge about the run-time behavior of the program. The compiler cannot completely predict run-time access patterns.

There is an exception to this rule. During the register allocation phase, the compiler often must insert substantial amounts of *spill code*; that is, instructions that move values from registers to memory and back again. Because the compiler itself inserts these memory instructions, it has more knowledge about them than other memory operations in the program.

Spill-code operations are disjoint from the memory manipulations required by the semantics of the program being compiled, and, indeed, the two can interfere in the cache. This paper proposes a hardware solution to the problem of increased spill costs—a small *compiler-controlled memory* (CCM) to hold spilled values. This small random-access memory can (and should) be placed in a distinct address space from the main memory hierarchy. The compiler can target spill instructions to use the CCM, moving most compiler-inserted memory traffic out of the pathway to main memory and eliminating any impact that those spill instructions would have on the state of the main memory hierarchy. Such memories already exist on some DSP microprocessors. Our techniques can be applied directly on those chips.

This paper presents two compiler-based methods to exploit such a memory, along with experimental results showing that speedups from using CCM may be sizable. It shows that using the register allocation's coloring paradigm to assign spilled values to memory can greatly reduce the amount of memory required by a program.

1 Introduction

To an outside observer, the DSP market appears to be a breeding ground for architectural innovations. These machines change quite rapidly; new features appear with each generation. The best of these features persist; some even make it into commodity microprocessors.

Some DSP chips have a small, fast, on-chip memory that the programmer can use to improve access times. These on-chip memories are not caches; instead, they are located in a disjoint address space. This simplifies their implementation by eliminating any need for relating on-chip memory addresses to off-chip memory addresses, for associative lookup, and for automatic replacement. Instead, the designers make the programmer responsible for moving data between main memory and the on-chip memory in a timely and efficient way. In the DSP arena, this strategy has worked.

This paper presents an alternative use for a small portion of this on-chip memory—as a holding place for spilled values, or a small *compiler-controlled memory* (CCM). Our scheme has several advantages. Using the CCM for spills should shorten spill latencies and let the scheduler place the load for a spilled value next to its use — speeding up execution and shortening the live range created for the spilled value. Spilling to the CCM removes spill traffic from the path to main memory. If the system has a cache memory, spilling to the CCM should also eliminate any cache pollution introduced by spill operations—loads and stores that can interfere directly with the cache behavior "planned" by high-level, compiler-based transformations that exploit locality caused by regular accesses in loop nests [8, 27, 10].

[†] This material is based, in part, upon work supported by the Texas Advanced Technology Program under grant number 003604-015 and by DARPA through USAFRL Contract F30602-97-2-298.
Authors' current e-mail addresses are cooper@cs.rice.edu and harv@cs.rice.edu.

This work is narrowly focused on spill-code insertion during register allocation. It has similarities to register promotion, a transformation that moves some pointer-based variables into registers [16]. Because of this similarity, we call the software component of this technique *spill promotion* – in both cases, the transformation moves a value's primary storage into a faster class of memory.

While we originally conceived of this scheme as a way to make automatic use of some portion of the local memory on a DSP chip, our experiments suggest that it may be worth considering a small CCM on commodity microprocessors, as well. To assess the reasonableness of this idea, we need to understand three issues: hardware support, software support, and the amount of CCM that would be needed.

This paper presents our vision for the CCM. Section 2 describes our assumptions about the underlying hardware, discusses why register spilling is problematic, and suggests why a CCM is a better solution than simply using the existing cache memory. Section 3 describes how to build a stand-alone CCM-allocator that runs as a post-pass to the compiler, and how to fit a CCM scheme into the spill-code insertion phase of a Chaitin-Briggs allocator. Section 4 presents experimental results that show the kind of improvements that should be expected from the use of a CCM and the amount of CCM required by programs in our test suite.

Finally, we must reiterate that our techniques should be directly applicable on some current DSP microprocessors. These chips already have small, fast, on-chip memories that are not caches; these systems expect the applications programmer to explicitly manage transfers into and out of these small, fast memories. On such systems, our results suggest a style of programming where the application programmer cedes the bottom 1 KB of on-chip memory to the compiler, which uses it to implement CCM-style spilling.

2 Background

Our proposal for CCM spilling combines both hardware and software issues. This section looks at the hardware requirements and explores the software and performance issues that motivate the proposal.

2.1 Hardware requirements

The primary hardware requirement is a small on-chip memory, the CCM. Conceptually, the CCM should have its own address space, accessible through designated instructions that move data between the CCM and the register set. Access to the CCM should be fast, with the results available for use on the next cycle. For each class of registers (*e.g.*, floating-point registers or general purpose registers), a pair of instructions will suffice:

$$\begin{array}{ll} \texttt{spill}\ r_j,\ \langle \textit{offset} \rangle & r_j \Rightarrow \text{CCM}[\langle \text{offset} \rangle] \\ \texttt{restore}\ \langle \textit{offset} \rangle,\ r_i & \text{CCM}[\langle \textit{offset} \rangle] \Rightarrow r_i \end{array}$$

Other applications for the CCM may find uses for more complex addressing modes. For our purposes, these simple absolute addresses are both sufficient and desirable.

If the compiler can assume that the system runs only one process, as with many embedded applications, the CCM can be small and simple—on the order of 1 KB.[1]

In a multi-tasked environment, the CCM should be larger and slightly more complex. To handle multiple processes, we would want to add a system-controlled base register to provide each process with its own small region within the CCM. This would allow the system to avoid copying the CCM contents to main memory on context switches. Still, we expect that a CCM of 16 KB to 32 KB would be more than adequate.

2.2 The importance of spilling

A modern microprocessor presents the compiler with a complex set of challenges. To achieve a reasonably large fraction of the processor's peak performance, the compiler must keep multiple, pipelined functional units busy, and arrange memory accesses in an order that creates good cache locality.

- To keep the pipelines busy, the compiler must ensure that an instruction is ready to execute on each functional unit at each cycle. This requires careful instruction scheduling. It also requires that the operands of each instruction be available, in registers, at the start of the appropriate cycle.
- To improve cache locality, the compiler must reorder and rearrange loop iterations. The "heroic" transformations that improve locality often introduce new overhead computations; they may also move some memory references into registers.

The transformations that address these problems can increase the demand for registers and provoke the register allocator to spill more values to memory. The perceived increase in the cost of spilling, due to more spills and longer memory latencies, has led to a spate of recent research on reducing spill costs (*e.g.*, [16, 3, 12]).

Register spills are problematic for two reasons. First, they add loads, stores, and address computations to the program, each of which must be scheduled, fetched, and executed. Second, they perturb the behavior of the data

[1] According to Upton *et al.*, the cost of a 1 KB cache is negligible [26]. The CCM should have a simpler implementation than a full-blown cache.

cache(s) and increase memory bandwidth requirements. Both effects are complicated by the fact that allocation occurs quite late in compilation; the transformations that block for cache and register locality cannot foresee the spills to account for their impact.

2.3 The cache is the wrong place to spill

Most modern microprocessors use cache memories as a tool to bridge the expanding gap between the speed of main memory and the speed of the processor [29]. At any point in time, a processor's cache holds a mapped subset of the address space that includes main memory. Hardware mechanisms adjust both the contents of cache and the mapping between cache locations and main memory locations as programs execute [13]. Cache systems, combined with compiler-based transformations, have been reasonably effective at bridging the gap between memory speed and processor speed [19]. Unfortunately, spill code inserted in the last stages of compilation can disrupt the compiler's carefully planned sequence of memory accesses.

From a performance perspective, the cache's importance lies in one simple fact: a memory reference to an element already in the cache takes much less time than a reference to an element not in the cache. A reference that "hits" the cache typically completes in a single cycle, while a reference that "misses" takes five to ten cycles on a simple uniprocessor machine, and as long as hundreds of cycles in a distributed memory multiprocessor [14, 15, 22, 17, 1, 2]. This difference in access time has a strong impact on the performance of individual programs. Accordingly, much recent research in compilation has been directed at techniques that improve the likelihood of references hitting in the cache. Most of this work falls into two major categories.[2]

Blocking Blocking rearranges and reorders the iterations of a loop nest in an attempt to move multiple references to a single location closer in time (*temporal locality*) or in an attempt to move references to adjacent memory locations closer in time (*spatial locality*) [27, 8, 28]. Blocking often introduces new loops with new index variables; this can increase the demand for registers and cause additional spilling.

Prefetching With appropriate hardware support, such as an *advisory prefetch instruction* [7], the compiler can hide the latency associated with a cache miss. To do this, it inserts a prefetch well in advance of the reference that it believes will miss the cache. In response, the hardware pre-loads the location into cache [21, 7, 10]. Prefetching creates new copies of address calculations; this either lengthens an existing live range or creates a new one. Both effects can cause additional spilling.

In both blocking and prefetching, the compiler analyzes the program and tries to predict the run-time behavior of the cache. The compiler then bases its transformations on those predictions. The analysis and prediction techniques rely on an implicit assumption that all accesses are exposed to the analysis; they also tend to focus on repeated references to arrays. When scalar references conflict with the cache behavior planned by the compiler, they reduce the impact of those transformations.

Further, McKinley and Temam studied the actual memory behavior of a suite of scientific programs [19]. They conclude that the data cache is not well used even for programs with relatively high hit rates [18]. In this study, nearly twenty-five percent of all memory references are scalar values rather than array values. While McKinley and Temam do not report the percentage of references that are inserted by the register allocator, we do know that every spill is a scalar reference.

Other papers have addressed the issue of reducing the cost of programmer-inserted scalar references [8, 9]. In this paper, we address the issue of reducing the cost of compiler-inserted scalar references. In some ways, these compiler-inserted references are more insidious. Since spills are inserted in the final stages of compilation, the earlier transformations cannot see them, analyze them, plan for them, or eliminate them.

Our proposal for a small, fast, CCM would eliminate much of the unplanned disruption caused by spill code. The rest of this paper focuses on the software support needed to utilize the CCM for spilling. We present two different ways that a compiler-writer could handle CCM spilling and present experimental evidence to demonstrate the potential impact of adopting this collaborative hardware/software scheme.

3 Software Implementation

The software support required to use the CCM for spilling can take several forms. We built two different implementations. The first is a post-pass CCM allocator that operates after traditional register allocation. The second embeds support for CCM spilling directly into a typical Chaitin-Briggs allocator [4]. Our experiments show that the two approaches produce different results.

---

[2] A third approach, *streaming*, was used on the *i*860. Here, the compiler would load values from memory to registers, bypassing the cache, and then write them into an array that modeled the cache. This gave the compiler fairly precise control over the contents of cache by largely avoiding instructions that could cause a replacement. The control came at the expense of doubling the amount of data movement [20].

Calculate the call graph (if necessary) [23, 6]
Conservatively mark subroutines in call-graph cycles as using all of CCM
For each subroutine, s, in a postorder walk over the call graph:
  Rewrite spill instructions with symbolic names
  Perform liveness analysis over spill locations
  Build SSA on the spill locations
  Build live-range names
  Build the interference graph
  Calculate the cost of each live range
  Allocate live ranges to CCM locations by coloring
  Rewrite spill instructions to spill to CCM
  Record the amount of the CCM used by s

Figure 1 Algorithm for post-pass spill promotion

3.1 A post-pass CCM allocator

The post-pass CCM allocator is invoked after the end of traditional compilation. It takes allocated, scheduled code as input and produces equivalent code where some of the loads and stores inserted for spilling have been redirected into the CCM. It uses the basic algorithms of the Chaitin-Briggs allocator [4], but operates over a somewhat different domain.

The CCM allocator focuses on spill locations rather than data values in registers. When it runs, all decisions about which live ranges stay in registers have been made. The CCM allocator tries to discover a subset of the spilled values that can be safely and profitably relocated to the CCM. Thus, instead of building SSA form for values as is typical, the CCM allocator builds an SSA form for the memory locations that hold spills. It uses addresses as symbolic names and builds the analog of live-range names from that base set of symbolic names.

In this context, the notion of "liveness" changes. A spill location, m, is "live" at some point, p, in a program if there exists an execution path from p to an instruction that loads m. Thus, m is live at p if it might be loaded again after p executes. A memory location m is "defined" when the program stores a value into m. It is "used" when the program loads a value from m. Using these new definitions, the CCM allocator computes the analog of liveness information for spill locations. The liveness information is used, in turn, to build an interference graph that shows whether or not two spill locations can share a single memory location (in main memory or in the CCM).

The CCM allocator does not generate new spills. Instead, it redirects some subset of the existing spills into a size-limited CCM. When it discovers that the spills for a particular value will not fit into the CCM, it simply leaves the original spill code intact. This causes the spills for that value to remain in main memory, producing a heavyweight spill rather than a CCM spill. The result is conservative, but safe.

Traditional register allocators are intraprocedural in nature—that is, they treat individual procedures as independent entities. When a traditional register allocator allocates memory to hold a spilled value, it typically places the spilled value in the current activation record of the procedure containing the spill. Thus, allocation decisions for spill locations can be made independently in different procedures. With the CCM, the situation changes. Since the CCM is a global resource, shared across the whole program, the CCM allocator must adopt some interprocedural conventions on the use of CCM.

Two distinct strategies make sense. First, the allocator can avoid the problem by limiting CCM spilling to values that are not live across any call site. These values will never be in the CCM when another procedure is active, so their use of the CCM cannot conflict with any other procedure. Second, the allocator can coordinate the use of CCM across procedures to avoid conflicts. This requires information about the use of CCM by other procedures in the program; this strategy makes the most sense in a post-pass CCM allocator that runs at link-time.

We implemented both strategies. In the absence of interprocedural information, the post pass allocator adopts the conservative strategy and only uses CCM for values that are not live across calls. If interprocedural information is available, it performs allocation in a bottom-up walk over the call graph. That is, it processes all routines reachable from procedure p before considering p. After it processes a procedure q, it records, for each call to q, the amount of CCM that q uses. When allocating for p, the allocator can then use any location for a value that is not live across the call, but must use a location higher in CCM than q's high-water mark for values that are live across the call to q.

To handle cycles in the call graph, corresponding to recursion, it behaves conservatively. It marks each procedure in the cycle as using the full CCM.

Figure 1 shows a sketch of the post-pass CCM allocation algorithm. After building the supporting data-flow analyses such as liveness and SSA for the spill memory positions, we build the interference graph. The coloring algorithm is quite similar to those used for register allo-

Loop until no new spill code is added:
    Build SSA Form (include CCM positions)
    Build live-range names
    Repeat until no more coalescing possible
        Build the interference graph (include CCM positions)
        Coalesce copies
    Calculate spill costs
    Simplify
    Select
    Spill (try to spill into CCM positions)

Figure 2 Modified register-allocation algorithm including spill promotion

cation, except that when the graph contains only nodes of high degree, we simply remove the cheapest from the graph, allowing it to remain as a heavyweight spill instruction, and proceed.[3] Note that, unlike the Chaitin-Briggs register allocator, the post-pass CCM allocator only needs a single pass of coloring for each subroutine.

To select a specific location in the CCM, the post-pass allocator uses the same algorithm that the Chaitin-Briggs allocator uses to pick a spill location in main memory. It starts at the beginning of the CCM and tries successive locations until it finds one that will work—that is, a location not used by any interference-graph neighbor of the spilled value. Here, the "beginning" of this search space is the maximum of the CCM usage in the set of subroutines across which the spilled value is live. We manage this issue by creating an array of integers that corresponds to spill locations; a pass over the code uses the live sets at subroutine calls to compute the appropriate "beginning" address for each spill location.

3.2 CCM allocation during spill-code insertion

To incorporate CCM spilling into the "normal" register allocator, we must make the CCM locations visible in the allocation process. The allocator assigns an abstract name to each CCM location and extends its data-flow analysis to include these CCM names. Thus, CCM locations appear in the interference graph alongside live ranges of program values. On the initial pass through the register allocator, the CCM locations have no interferences. At the end of the first allocation pass, the act of inserting spill code that uses CCM locations will create spans over which they are live. This, in turn, forces edges between CCM locations and live ranges into the interference graph.

The allocator ignores these edges during allocation and uses them during spill code insertion. When it goes to spill a value, edges between the node and CCM locations show the allocator which CCM locations cannot hold the spilled value. This simplifies the search for an appropriate CCM location.[4]

Managing name spaces is one of the more difficult parts of building a Chaitin-Briggs allocator. By comparison to the problems introduced by building SSA, such as forming live ranges, coalescing live ranges, and coloring the result into a small space of register and spill location names, adding a name space for CCM locations was relatively easy.

1. CCM names are introduced after live ranges have been discovered. The allocator appends the set of CCM names onto the set of live ranges. This occurs just before construction of the interference graph.
2. Interference graph construction treats CCM location names in the way that it handles register values. It uses the definition of liveness cited in the last section: a CCM location becomes live when it is stored to, and it remains live until the last load from that position. With that definition, the only real change to the interference-graph building algorithm is insertion of code which recognizes spills to the CCM – CCM locations are otherwise treated exactly as register values.
3. After building the interference graph, CCM locations are ignored until the *spill* section of the allocator. Each time a value is marked for spilling under normal register allocation, the allocator assigns it to a spill location, usually based on the value of the stack pointer. We modify this by inserting a check to see if the value can be assigned to the CCM instead of being put on the stack.
4. During spill insertion, the allocator consults the interference graph to determine whether or not a suitable CCM location is available. The governing rule is: a value v cannot be spilled to CCM position m if an edge from v to m is in the interference graph. As CCM spills are inserted, we must update the interference graph. Spilling v to

[3] Any remaining heavyweight spill instructions should have their spill locations updated so that they are packed tightly together and so use the least memory necessary; indeed, the interference graph already built can be used to compact the memory.

[4] Our allocator does not color spills to main memory, so the equivalent edges are not present for main memory spill locations. If the allocator runs out of spill locations in main memory, it simply extends the activation record for the current procedure. Since the CCM is a fixed-size resource, the more expensive approach is warranted.

Routine	Bytes of Spill Memory			*Routine*	Bytes of Spill Memory		
Name	*Before*	*After*	$\frac{After}{Before}$	*Name*	*Before*	*After*	$\frac{After}{Before}$
twldrv	12,024	10,376	0.86	saturr	976	520	0.53
fpppp	9,712	2,976	0.31	radb3X	928	456	0.49
deseco	6,536	5,824	0.89	radf3X	912	456	0.50
erhs	4,512	4,136	0.92	smoothX	760	432	0.57
fieldX	3,856	1,416	0.37	advbndX	736	528	0.72
jacld	3,728	3,536	0.95	radb2X	688	376	0.55
rhs	3,544	2,888	0.81	ddeflu	688	496	0.72
parmvrX	2,656	2,472	0.93	radf2X	680	368	0.54
jacu	2,368	2,328	0.98	vslv1pX	640	368	0.57
radbgX	2,296	976	0.43	vslv1xX	488	344	0.70
radfgX	2,112	744	0.35	efill	480	472	0.98
supp	1,584	824	0.52	colbur	432	416	0.96
radb5X	1,472	704	0.48	svd	408	304	0.75
radf5X	1,456	696	0.48	tomcatv	376	368	0.98
radf4X	1,328	640	0.48	dyeh	360	184	0.51
radb4X	1,320	648	0.49	getbX	288	256	0.89
subb	1,312	672	0.51	putbX	272	240	0.88
parmovX	1,168	1,024	0.88	parmveX	264	200	0.76
				cosqf1X	232	224	0.97
TOTAL	73,592	49,888	0.68				

Table 1 Spill Memory Requirements and Compaction

CCM location m requires an edge from m to each live-range neighbor of v in the interference graph.[5]

The expanded spilling algorithm is shown in Figure 2. This is the algorithm described by Briggs [4]; we embolden each step which requires modification.

4 Experimental Results

To evaluate these CCM allocation techniques, we implemented three distinct algorithms: the post-pass CCM allocator in both intraprocedural and interprocedural versions, and a modification of the Chaitin-Briggs allocator that directly generates CCM spills. We also built a memory compaction routine that colors spill memory to make non-interfering spilled values occupy the same memory location when possible.

We ran these algorithms on a suite of 122 routines, drawn from sources that include code from Forsythe *et al.*'s book on numerical methods [11], the SPEC '89 benchmarks [24], and the SPEC '95 benchmarks [25]. Out of this suite, 59 routines required some amount of spill code, and it is on these 59 routines that the following numbers were generated. All the routines were subjected to extensive scalar optimization, including global value numbering, global constant propagation, global dead-code elimination, partial redundancy elimination, and peephole optimization. Further, the routines in the `wave` benchmark were subjected to loop transformations to enable prefetching analysis; these transformations have the effect of greatly increasing the register pressure, and affected routines have an 'X' suffix added to the name.

The test codes in our suite are all Fortran-derived ILOC, the intermediate code of our compiler [5]. ILOC is a low level, three-address intermediate representation similar to assembly code. The back-end is a translator from ILOC to heavily instrumented C code, which we then compile with Sun's `cc` compiler. This back-end design allows us to modify the parameters of the abstract target machine while instrumenting the code to give us valuable insights into the execution of the code.

We compiled these codes to an extremely simple machine model. It has 64 registers (32 general-purpose registers and 32 floating-point registers). It issues a single instruction per cycle. We assumed that memory operations cost two cycles and all other instructions (including accesses to the CCM) execute in a single cycle. With the exception of the single-issue rule, this model roughly approximates the simpler DSP chips that we have been investigating. The cycle counts used in this section come from counters in the instrumented C code.

4.1 Spill memory size

The first question that we wanted to answer was "how much CCM is necessary?" To discover the answer, we ran the standard Chaitin-Briggs register allocator, followed by a coloring-based memory compaction scheme. Of the 59 routines that require spill code, coloring spill memory reduced the memory requirements of thirty-seven

[5] Alternatively, the allocator can keep a side data structure of the CCM spills inserted in the current round of spilling. Then, the rule becomes, v cannot be spilled to m if $\langle v, m\rangle$ is in the interference graph, or there is an edge $\langle v, p\rangle$ in the graph and p has already been spilled to m in this round of spilling.

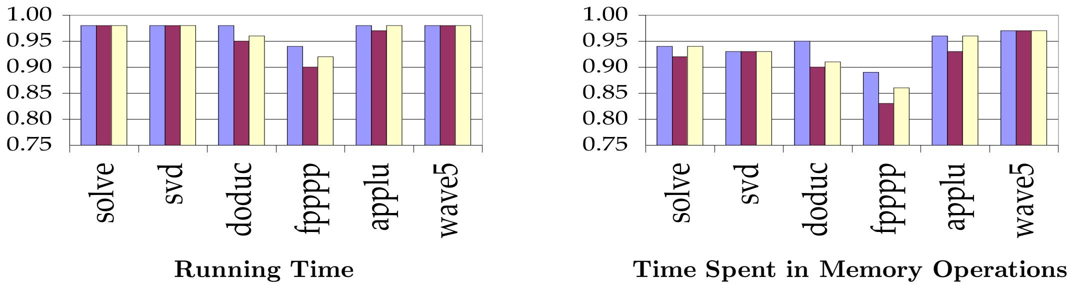


Figure 3 Program Performance with a 512-byte CCM

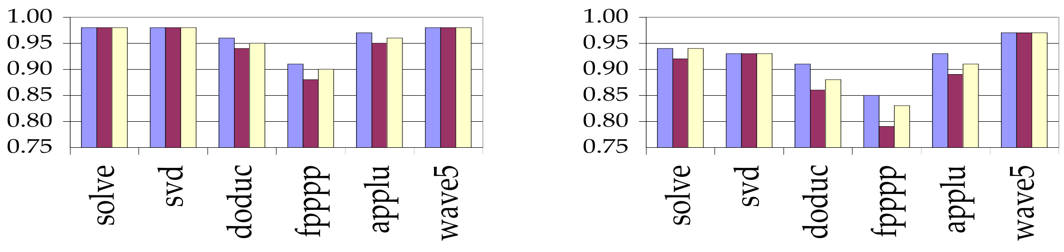
Routine Name	Without CCM	Post-Pass	Post-Pass w/ Call Graph	Integrated
decomp	825(252)	0.97(0.90)	0.96(0.88)	0.97(0.90)
svd	5,375(1,740)	0.98(0.93)	0.98(0.93)	0.98(0.93)
saturr	26,432(17,360)	0.90(0.84)	0.85(0.77)	0.90(0.84)
subb	959,280(546,000)	0.92(0.86)	0.84(0.72)	0.84(0.72)
supp	1,288,056(744,464)	0.93(0.88)	0.86(0.76)	0.86(0.77)
colbur	321,659(135,408)	0.99(0.98)	0.99(0.98)	0.99(0.98)
debflu	328,160(153,888)	1.00(1.00)	1.00(0.99)	1.00(1.00)
bilan	194,218(86,790)	1.00(0.99)	1.00(0.99)	1.00(0.99)
ddeflu	579,443(254,760)	0.99(0.98)	0.92(0.82)	0.99(0.98)
deseco	970,072(470,140)	0.97(0.94)	0.96(0.92)	0.97(0.94)
pastem	223,961(53,308)	1.00(0.99)	1.00(0.99)	1.00(0.99)
efill	4,705,095(2,570,138)	0.78(0.59)	0.78(0.59)	0.78(0.59)
fpppp	57,782,160(37,234,728)	0.95(0.92)	0.89(0.83)	0.91(0.86)
twldrv	111,128,482(60,658,016)	0.93(0.88)	0.91(0.83)	0.91(0.83)
setiv	236,731(75,692)	0.99(0.96)	0.99(0.96)	0.99(0.96)
erhs	2,873,175(1,223,036)	0.99(0.98)	0.99(0.98)	0.99(0.98)
rhs	78,222,321(38,056,812)	0.98(0.97)	0.98(0.97)	0.98(0.97)
jacld	58,617,550(34,801,900)	0.95(0.92)	0.90(0.83)	0.96(0.94)
blts	108,885,800(45,403,600)	0.99(0.98)	0.99(0.98)	1.00(0.98)
jacu	40,553,050(23,788,900)	0.93(0.88)	0.86(0.76)	0.93(0.88)
buts	109,578,600(46,621,300)	1.00(0.99)	1.00(0.99)	1.00(0.99)
denptX	5,940,008(2,070,008)	0.98(0.96)	0.98(0.96)	0.98(0.96)
parmvrX	261,706,085(152,279,892)	0.99(0.99)	0.99(0.99)	0.99(0.99)
smoothX	33,683,283(21,466,402)	0.97(0.95)	0.97(0.95)	0.97(0.95)
rffti1X	712(188)	0.97(0.87)	0.97(0.87)	0.97(0.87)
radf5X	3,803,650(2,159,000)	0.99(0.98)	0.86(0.76)	0.86(0.76)
radf4X	5,718,175(3,613,150)	0.84(0.74)	0.79(0.67)	0.79(0.67)
radf2X	3,362,325(2,000,250)	0.85(0.74)	0.85(0.74)	0.85(0.74)
vslv1pX	21,803,385(15,174,540)	0.97(0.96)	0.97(0.96)	0.97(0.96)
radb2X	3,406,775(2,038,350)	0.84(0.74)	0.84(0.74)	0.85(0.74)
radb4X	5,689,600(3,632,200)	0.84(0.75)	0.79(0.67)	0.79(0.67)
radb5X	3,667,125(1,905,000)	0.99(0.98)	0.89(0.80)	0.89(0.80)
slv2xyX	49,928(35,492)	1.00(1.00)	0.99(0.99)	1.00(1.00)
fieldX	12,172,573(6,821,302)	0.93(0.88)	0.93(0.87)	0.93(0.87)
initX	8,877,019(3,071,802)	1.00(1.00)	0.99(0.98)	1.00(1.00)

Table 2 Speedups in Dynamic cycle counts with 512-byte CCM

routines. Table 1 shows the results for these thirty-seven routines. Each entry shows the number of bytes of spill memory required before and after compaction and the ratio $\frac{\text{after}}{\text{before}}$. Of the 22 routines for which no compaction was possible, only four of them, `paroi`, `inisla`, `energyX`, and `pdiagX`, required more than 1000 bytes of spill memory.

4.2 Execution-time improvements

Based on our measurements of spill memory requirements, we chose a one kilobyte CCM as the target for our experiments. This size accommodates three quarters of the subroutines. We also ran the experiments with half that amount of memory, to see if key spill promotions would be responsible for the majority of the speedup. This gives us two data points. First, it shows the rela-



Running Time **Time Spent in Memory Operations**

Figure 4 Program Performance with a 1024-byte CCM

Routine Name	Without CCM	Post-Pass	Post-Pass w/ Call Graph	Integrated
subb	959,280(546,000)	0.84(0.72)	0.80(0.65)	0.80(0.65)
supp	1,288,056(744,464)	0.86(0.77)	0.80(0.66)	0.80(0.66)
prophy	378,156(209,264)	1.00(1.00)	0.90(0.81)	1.00(1.00)
fpppp	57,782,160(37,234,728)	0.92(0.87)	0.82(0.72)	0.86(0.79)
twldrv	111,128,482(60,658,016)	0.91(0.83)	0.91(0.83)	0.91(0.83)
jacld	58,617,550(34,801,900)	0.91(0.85)	0.82(0.69)	0.87(0.78)
jacu	40,553,050(23,788,900)	0.86(0.76)	0.82(0.69)	0.82(0.69)
radf5X	3,803,650(2,159,000)	0.86(0.76)	0.86(0.76)	0.86(0.76)
radf4X	5,718,175(3,613,150)	0.79(0.67)	0.79(0.67)	0.79(0.67)
fieldX	12,172,573(6,821,302)	0.93(0.87)	0.93(0.87)	0.93(0.87)
initX	8,877,019(3,071,802)	1.00(1.00)	0.99(0.98)	1.00(1.00)

Table 3 Changes in speedups in dynamic cycle counts with 1024-byte CCM compared to a 512-byte CCM

tive speedup and the cost/benefit of having more CCM. Second, it shows that even with a CCM of 512-bytes, significant speedups can be achieved.

The results of running these codes on our simple abstract machine are shown in Tables 2 and 3. The first column shows the dynamic execution costs, in cycles, of each routine. The parenthesized number indicates the number of cycles used for memory operations. The second column shows the relative number of cycles executed by the code after running the post-pass allocator with no interprocedural information, and, again, the number in parentheses is the relative reduction in cycles spent on memory operations. The third column shows the relative reduction of cycles spent for codes run through the post-pass allocator with interprocedural information. The final column shows the relative reduction of cycles used by codes run through the register allocator with CCM allocation built into the spill-code generator. To emphasize that the change in dynamic execution did not change very much when we increased the CCM size from 512 bytes to 1024 bytes, Table 3 only reports on routines which sped up as a result of using a larger CCM.

Figures 3 and 4 show total running times for the six programs (out of 13) which showed improvement. Each program has three bars associated with it, corresponding to the three different CCM allocation methods, intraprocedural post-pass, interprocedural post-pass, and the integrated allocator. The values shown are the dynamic execution times relative to running the programs without CCM.

Table refaverages table shows the weighted average reduction in both total cycles executed and cycles spent on memory operations. Clearly, having interprocedural information offers a small improvement to the results, and we might expect a register allocator with such analysis at its disposal to perform even better than the three algorithms presented herein.

4.3 More complex execution models

Of course, our simple execution model does not capture all of the complex behavior of a general-purpose microprocessor. In particular, a more complex memory hierarchy will produce different results. Some changes in the memory hierarchy should achieve effects similar to CCM, while others might make CCM more attractive.

Better Cache Using a bigger or more effective cache should decrease the impact of spills on cache behavior. It should also make spills through the cache faster, on average. However, it leaves all the spill traffic on the pathway to main memory. We expect CCM would be less costly since it has a much simpler interface.

	Percentage Reduction in			
	Total Cycles Executed		*Cycles Spent In Memory Operations*	
Algorithm	*512-byte* CCM	*1024-byte* CCM	*512-byte* CCM	*1024-byte* CCM
Post-pass	3	4	10	13
Post-pass w/ Call Graph	4	6	14	17
Integrated	3	5	11	15

Table 4 Weighted-average reduction in cycles executed for each algorithm

Write Buffer Adding a write buffer should alleviate the cache effects of stores generated by spilling. It does little or nothing for loads generated by spilling. Most spilled values are re-loaded; in fact the detailed numbers in places like Briggs' thesis show that spilling causes many more loads than stores [4].

Victim Cache Adding a victim cache would probably alleviate the ill effects of spill-induced cache replacements. Using a victim cache rather than CCM leaves the spill traffic on the pathway to main memory. Again, the implementation of a victim cache might be more expensive than a small CCM.

Prefetching The impact of advisory prefetch is hard to predict. Like CCM, it requires modifications in the compiler to generate the appropriate instructions. If done successfully, prefetching should lower the average cache miss penalty for access to variables. Unfortunately, spill code is added much later in compilation; thus, the deleterious effects of spilling might actually be more pronounced with prefetching. This might make CCM look more attractive.

Finally, we declined to consider the effects of scheduling, which can simultaneously hide the memory latencies and cause added spilling due to increased register pressure.

5 Summary and Conclusions

In this paper, we proposed the creation of a small, fast, on-chip memory that the compiler could use for scalar memory operations inserted by the register allocator and showed the minor software enhancements necessary to utilize this memory space. Our proposal offers the opportunity to move scalar spill operations out of the address space containing both cache and main memory. This eliminates cache pollution introduced by spill instructions and reduces total traffic through the cache.

The hardware requirements for our CCM are simple; the amount of CCM required for real programs is quite modest. Indeed, using only a small amount of CCM (512 to 1024 bytes) produced significant decreases in the run-time cost of register spilling. Because the amount of memory per program is so small, it might be worth considering implementing future general-purpose microprocessors with a CCM with 16KB to 32KB of memory and adding a simple mechanism for context-switching it without copying it back to main memory.

We showed how to incorporate the necessary compiler support in two ways: as a post-pass CCM allocator that could run at the end of compilation or at link-time, or as an integrated part of a Chaitin-Briggs allocator.

This idea may find immediate application on some of the current generation of DSP chips. On DSP chips with a small local memory, reserving the bottom 512 to 1024 bytes of that memory would allow the compiler to apply the techniques presented here. Many of these chips are used in applications that run but a single process; in that environment, context switching is not an issue.

Acknowledgments

This work was supported by DARPA through USAFRL Contract F30602-97-2-298, and by the Texas Advanced Technology Program under grant number 003604-015. This work would not have been possible without much help over the years from the group of people who have worked on the Massively Scalar Compiler Project. We owe a particular debt of thanks to Kathryn McKinley, John Bennett, Preston Briggs, John Pieper, Linda Torczon, Parthasarathy Ranganathan, and the anonymous reviewers for their help and their advice.

References

[1] Anonymous. Performance of pentium pro and pentium ii processor/cache combinations. Technical report, ECG Technology Communications Group, Compaq Computer Corporation, May 1997.

[2] Bary R. Beck, David W.L. Yen, and Thomas L. Anderson. The cydra 5 minisupercomputer: Architecture and implementation. *The Journal of Supercomputing*, 7, 1993.

[3] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew O'Keefe. Spill code minimization via interference region spilling. *SIGPLAN Notices*, 32(6):287–295, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.

[4] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[5] Preston Briggs. The massively scalar compiler project. Technical report, Rice University, July 1994. Preliminary version available via anonymous ftp.

[6] David Callahan, Alan Carle, Mary W. Hall, and Ken Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4), April 1990.

[7] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, 1991.

[8] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1994.

[9] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu. A new algorithm for partial redundancy elimination based on ssa form. *SIGPLAN Notices*, 32(6):273–286, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[10] Keith Cooper, Ken Kennedy, and Nathaniel McIntosh. Cross-loop reuse analysis and its application to cache optimization. In Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing, San Jose, California, 1996.

[11] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[12] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3):300–324, May 1996.

[13] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1990.

[14] Cristina Hristea, Daniel Lenoski, and John Keen. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, pages ??–??, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.

[15] Intel Corporation. *Pentium™ II Processor Developer's Manual*, 1997.

[16] John Lu and Keith Cooper. Register promotion in c programs. *SIGPLAN Notices*, 32(6):308–319, June 1997. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation.*

[17] Sally A. McKee. Compiling for efficient memory utilization. In *Workshop on Interaction Between Compilers and Computer Architectures, Second IEEE Symposium on High Performance Computer Architecture (HPCA-2)*, San Jose, California, January 1996.

[18] Kathryn S. McKinley. Personal communication. Email message, July 1998.

[19] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1996.

[20] Larry Meadows, Steven Nakamoto, and Vincent Schuster. A vectorizing, software pipelining compiler for LIW and superscalar architecture. In *Proceedings of RISC '92*, San Jose, CA, February 1992.

[21] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, Massachusetts, 1992.

[22] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An evaluation of memory consistency models for shared-memory systems with ilp processors. In Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1996.

[23] Barbara G. Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, 5(3):217–226, May 1979.

[24] SPEC release 1.2, September 1989. Standards Performance Evaluation Corporation.

[25] SPEC release 1.10, September 1995. Standards Performance Evaluation Corporation.

[26] Michael Upton, Thomas Huff, Trevor Mudge, and Richard Brown. Resource allocation in a high clock rate microprocessor. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, 1994.

[27] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. *SIGPLAN Notices*, 26(6):30–44, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[28] Michael Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, Nevada, November 1989.

[29] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *Computer Architecture News*, 23(1), March 1995.