Latency-hiding and Optimizations of the DSZOOM Instrumentation System

Oskar Grenholm, Zoran Radović, and Erik Hagersten
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden

Abstract

An efficient and robust instrumentation tool (or compiler support) is necessary for an efficient implementation of fine-grain software-based shared memory systems (SW-DSMs). The DSZOOM system, developed by the Uppsala Architecture Research Team (UART) at Uppsala University, is a sequentially consistent fine-grained SW-DSM originally developed using Executable Editing Library (EEL)—a binary modification tool from University of Wisconsin-Madison. In this paper, we identify several weaknesses of this original approach and present a new and simple tool for assembler instrumentation: SPARC Assembler Instrumentation Tool (SAIT). This tool can instrument (modify) a highly optimized assembler output from the compiler for the newest UltraSPARC processors. Currently, the focus of the tool is load-, store-, and load-store-instrumentation.

By using the SAIT, we develop and present several low-level instrumentation optimization techniques that significantly improve the performance of the original DSZOOM system. One of the presented techniques is a *write permission cache* (WPC), a latency-hiding mechanism for memory-store operations that can lower the instrumentation overheads for some applications (as much as 45% for LU-cont, running on two nodes with eight processors each).

Finally, we demonstrate that this new DSZOOM system executes faster than the old one for all 13 applications studied, from the SPLASH-2 benchmark suite. Execution time improvement factors range from 1.07 to 2.82 (average 1.73).

1 Introduction

The DSZOOM-WF implementation [16], is a sequentially consistent [12], fine-grained distributed software-based shared memory system, implemented on top of the 2-node Sun WildFire [8, 9] prototype, without relying on its hardware-based coherence capabilities. All loads and stores are instead performed to the node's local "private" memory. An unmodified version of the executable editing library (EEL) [13] is used to insert fine-grain *access control checks* before shared-memory loads and stores in a fully compiled and linked executable. Global coherence is resolved by coherence protocols (which are triggered by the inserted access control checks) implemented in C, that copies data to the node's local memory with UltraSPARC processor's Block Load/Store operations to the remote memory (which is locally mapped in every node). The all-software protocol is implemented assuming some basic low-level primitives in the cluster interconnect and an operating system bypass functionality, similar to the emerging InfiniBand standard [10]. All interrupt- and/or poll-based asynchronous protocol processing, usually found in almost all traditional SW-DSM proposals so far [11, 14, 19, 18, 20, 23, 24], is completely removed by running the entire coherence protocol in the requesting processor.

Program *instrumentation* is a general term for techniques used to modify existing programs in order to typically collect the data during a program execution (tracing, cache simulation, profile information, etc.). During the program execution, the instrumentation code is executed together

with the original program code. The code instrumentation can typically be done at several different levels: hardware, library interposing, source code, assembler code, or machine level (i.e., binary instrumentation). Currently, in the DSZOOM system, the binary instrumentation is used [13], a technique that was also used in several similar projects in the past (e.g., Blizzard-S [21], Shasta [19, 18, 17], Sirocco-S [20]).

As mentioned above, the DSZOOM's instrumentation system is developed with EEL, which is unfortunately not maintained anymore. The current state of the EEL library makes it unusable for binaries that are compiled with high optimization levels and new compilers on modern operating systems. EEL is also unable to instrument all types of instructions that are placed in SPARC's delay slots. This is the main motivation for this work. To avoid those limitations we instead instrument the assembler output from the compiler, and insert the *snippets* (small fragments of assembler/machine code) needed. The compiler finishes its job of making it all into an executable. By doing the actual instrumentation at the assembler level, we can easily analyze and re-arrange the code in a way that eliminates loads/stores in delay slots. The problem of inserting code snippets is now reduced to inserting correct assembler code, as text, into a text file containing the assembler output of the program.

In this paper, we present a new instrumentation tool, SPARC Assembler Instrumentation Tool (SAIT), which is a simple and efficient instrumentation technique to instrument the assembler output from the compiler. The SAIT can instrument a highly optimized assembler output from Sun's latest compilers for the newest UltraSPARC processors. Currently, the focus of the tool is load-, store-, and load-store-instrumentation because this is of the biggest importance for the DSZOOM system. (The tool can easily be extended to support other types of instrumentation as well.) The major limitation of this approach is that the source code must be available, which is not always the case, especially not for the system libraries. On the other hand, instrumenting libraries with EEL or some other binary instrumentation tool is in practice very difficult (on some architectures even impossible) task since the code in libraries is usually heavily optimized and it can contain "data in code" and "code in data" segments that are very difficult to resolve without some additional help from the compilers.

We present several low-level instrumentation optimization techniques that significantly improve the performance of the original DSZOOM system. All new instrumentation techniques are developed with SAIT. One of the presented techniques is a *write permission cache* (WPC), a latency-hiding mechanism for memory-store operations. By using the WPC, the instrumentation overheads for some applications can be lowered (as much as 45% for LU-cont, running on two nodes with eight processors each). We also demonstrate that this new DSZOOM system executes faster than the old one for all 13 applications studied, from the SPLASH-2 benchmark suite. Execution time improvement factors range from 1.07 to 2.82 (average 1.73).

The rest of this paper is organized as follows. In section 2, we give a short target architecture/compiler overview. The SAIT is introduced in the section 3. Several low-level instrumentation optimization techniques, including the WPC, are described in section 4. In section 5, the performance study is performed, and finally, we conclude in section 6.

2 Target Architecture/Compiler Overview

2.1 Original Proof-of-Concept Platform

The system used to host the original proof-of-concept DSZOOM was a 2-node Sun WildFire [8, 15, 9], with Sun Enterprise E6000 SMP processing nodes [22], with 16 UltraSPARC II (250 MHz) processors per node, running a slightly modified version of the Solaris 2.6 operating system. The compiler used to compile both EEL and the SPLASH-2 benchmark programs was GNU's gcc-2.8.1. The benchmark programs were compiled without any optimization because EEL could

not always instrument the binaries produced otherwise. The proof-of-concept implementation was tested thoroughly on this platform. For further information on this original implementation using EEL, see [16].

2.2 SPARC V8 and V9 ABI Restrictions

The instrumentation process must be compliant with all SPARC ABI specifications, and especially with global register usage [25]. Currently, the DSZOOM engine requires two free global registers, at the insertion point during the instrumentation phase, to pass parameters to the coherence routines in an efficient way from in-line code snippets. On SPARC V8 (32-bit) and SPARC V8plus (64-bit) there are three global thread-*private* registers that are saved/restored during the thread-switching by the Solaris system libraries: %g2, %g3, and %g4; all other global registers are thread-*global* and are not saved during the switch. The thread-private registers are also called for *application registers*. On SPARC V9 (64-bit), on the other hand, only %g2 and %g3 are application registers, and the %g4 register is free for general use and is volatile across function calls together with %g1 and %g5. On all targets, registers %g6 and %g7 are reserved for system software and are not used during the code instrumentation process.

As mentioned above, we need two registers to pass arguments. For that purpose, we choose to use registers %g3 and %g4. On SPARC V8 or V8plus this choice leave us one extra register to use: %g2. This register is mainly used to make snippets a bit more efficient, and also for the write permission cache implementation (see section 4 for details). The DSZOOM should be possible to implement on SPARC V9 architecture as well, since usually only two application registers are used in snippets.

2.3 Target Compiler Details

The compiler used to produce the assembler output and the executables in this paper is Sun WorkShop 6 update 2 C 5.3 Patch 111679-08.[1] A brief overview of several relevant compiler flags, taken from the Forte Developer 6 update 2 manual [26], is given here.

-S Directs compiler to produce an assembly source file but not to assemble the program.

-xregs=r[,r...] Specifies the usage of registers for the generated code. *r* is a comma-separated list that consists of one or more of the following: [*no%*]*appl*, [*no%*]*float*. The *-xregs* values available are:

appl: Allows the use of the following registers: g2, g3, g4 (v8a, v8, v8plus, v8plusa, v8plusb) g2, g3 (v9, v9a, v9b). In the SPARC ABI, these registers are described as application registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.

no%appl: Does not use the appl registers.

-xO[1|2|3|4|5] Optimizes the object code; note the upper-case letter *O*. The levels (1, 2, 3, 4, or 5) you can use with *-xO* are described below.

-xO1 Does basic local optimization (peephole).

-xO2 Does basic local and global optimization. This is induction variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination, and complex expression expansion. The *-xO2*

[1] Today, the SAIT can also instrument the assembler output from Sun's Forte Developer 7 compilers (C, C++, and Fortran, version 5.4). These compilers are also called for Sun ONE Studio compiler collection.

level does not assign global, external, or indirect references or definitions to registers. It treats these references and definitions as if they were declared volatile. In general, the *-xO2* level results in minimum code size.

-xO3 Performs like *-xO2*, but also optimizes references or definitions for external variables. Loop unrolling and software pipelining are also performed. This level does not trace the effects of pointer assignments. When compiling either device drivers, or programs that modify external variables from within signal handlers, you may need to use the volatile type qualifier to protect the object from optimization. In general, the *-xO3* level results in increased code size.

-xO4 Performs like *-xO3*, but also automatically inlines functions contained in the same file; this usually improves execution speed. If you want to control which functions are inlined, see *-xinline=list*. This level traces the effects of pointer assignments, and usually results in increased code size.

-xO5 Attempts to generate the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback. See *-xprofile=p*.

-fast Selects the optimum combination of compilation options for speed. This should provide close to the maximum performance for most realistic applications. Modules compiled with fast must also be linked with fast. The fast option is unsuitable for programs intended to run on a different target than the compilation machine. In such cases, follow *-fast* with the appropriate *-xtarget* option. The fast option is unsuitable for programs that require strict conformance to the IEEE 754 Standard. The following table lists the set of options selected by *-fast* on the SPARC platform.

-dalign, -fns, -fsimple=2, -fsingle, -ftrap=%none, -xarch,
-xbuiltin=%all, -xlibmil, -xtarget=native, -xO5

fast acts like a macro expansion on the command line. Therefore, you can override the optimization level and code generation option aspects by following *-fast* with the desired optimization level or code generation option. Compiling with the *-fast -xO4* pair is like compiling with the *-xO2 -xO4* pair. The latter specification takes precedence. You can usually improve performance for most programs with this option. Do not use this option for programs that depend on IEEE standard exception handling; you can get different numerical results, premature program termination, or unexpected SIGFPE signals.

3 SAIT: SPARC Assembler Instrumentation Tool

The SAIT is first of all created for use with the DSZOOM system. It is a SPARC assembler parser with ability to insert code snippets at specified locations. Currently, the SAIT instruments integer-loads, floating-point-loads, and any type of stores. Java Compiler Compiler (JavaCC) version 2.1 is used for the design of SAIT's parser [27]. JavaCC is a parser generator for use with mainly Java applications. It works much like the classical tool, yacc, for the C programming language. With JavaCC it is also possible to write functions and additional code in Java, that can decide what to do with the parsed code (tree building is also possible to perform via a tool called JJTree, which is included with JavaCC). In this section, we shortly describe the main features/phases of the SAIT. For a more detailed description of the tool, see [7].

3.1 Parsing SPARC Assembler

Parsing the SPARC assembler is a quite simple task. An example of how the tokens for stores and registers are written in JavaCC, is given in Figure 1. All tokens are constructed from regular expressions.

```
< STORE: ("st" | "stb" | "sth" | "std" | "stx") >

< REG: "%" ["r","g","i","o","l","f","s","y"] (["0"-"9","p","o","c"])* >
```

Figure 1: Two examples of tokens in JavaCC.

The internal data structures of the tool consist of two different classes, implemented in Java, called Basic Block (BB) and Control Flow Graph (CFG). The formal definitions of basic blocks and control flow graphs is shown below [3].

Definition 1 A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

Definition 2 A control flow graph $G = (N; E; h)$ for a program P is a connected, directed graph, that satisfies the following conditions:

- h is the unique entry node to the graph,
- $\forall n \in N$; n represents a basic blocks of P, and
- $\forall e = (n_i; n_j) \in E$; e represents flow of control from basic block n_i to basic block n_j, and $n_i; n_j \in N$.

A basic block is just a collection of individual instructions. The set of instructions is always entered at the beginning and exited at the end. This means that they start with a label and ends with a branch, jump or call, or ends with the appearance of a new label. The algorithm for dividing a long sequence of statements into basic blocks is quite simple. The parser just looks for a label, then creates a basic block instance and adds all the following instructions into the structure, until a branch or a new label is found, then a new instance of a basic block is created and the old one is ended, and so on. This procedure is done once for every function block. At the end of a basic block, there is some information stored about where the next basic block that can be executed is, so that we can now know how the flow of the program will go. There are several different ways that a basic block usually can end [3]:

one-way the last instruction in the basic block is an unconditional jump to a label, hence, the block has one out-edge.

two-way the last instruction is a conditional jump to another label, thus, the block has two out-edges.

call the last instruction is a call to a procedure. There are two out-edges from this block: one to the instruction following the procedure call, and the other to the procedure that is called. Throughout analyses, the called procedure is normally not followed, unless inter-procedural analysis is required.

return the last instruction is a procedure return instruction. There are no out-edges from this basic block.

fall the next instruction is the target address of a branch instruction (i.e., the next instruction has a label). This node is seen as a node that falls through the next one, thus, there is only one out-edge.

Our model differs slightly from those definitions. In our case, we do not do any distinction between conditional and unconditional branches; both are two-way. In both cases, we just store the label that the branch is to, and the label that comes directly after the store. (The instruction in the delay slot belongs to the basic block as well.) Moreover, since we do not try to do inter-procedural analysis, we do not let our basic blocks end with a call, we just treat them as any other instruction. When the basic block just falls through, the label of the next basic block is stored. Finally, if the basic block ends with a return statement of some kind that indicates that the flow for this function or program ends here and accordingly information about this is stored instead.

After the creation of all the basic blocks, the flow of control is analyzed and a CFG is set up. A CFG is simply just a tree list over the different execution paths of the program. One CFG is constructed for each function in the program. Each node in this tree has a basic block and pointers to the next nodes (one or two pointers depending on if the exiting point is a straight code or a branch) or a *null* pointer if it is a terminating point.

3.2 Liveness Analysis

The SAIT needs to analyze the intermediate-representation structure to determine which registers are in use at the insertion point during the instrumentation phase. A register is *live* if it holds a value that may be needed in the future, so this analysis is usually called *liveness* analysis.

Armed with the information in the CFGs and the basic blocks, we can calculate the liveness of the registers in each basic block. Liveness is the information about which of the registers that holds values to be used later on in the program and which that can be overwritten without altering the execution of the program. To be able to do this, we need to know exactly which registers each instructions uses and defines (defs). To clarify, an assignment to a register defines that register and an occurrence of a register on the right-hand side of an assignment (or in other expressions) uses that register. For example in the following SPARC assembly statement, `add %o1,%o2,%o3`, registers %o1 and %o2 are used, and %o3 is defined. Knowing the different execution paths of a function, and which registers each basic block *uses* and *defs*, there is a simple algorithm for calculating the liveness at each basic block. This algorithm is given in Figure 2. The algorithm returns a hash table for each basic block in every CFG that holds those registers that are live at the entering point and at the exit point of the block. Free registers (non-*live*) are not included in the table.

Terminology. A flow-graph node has *out-edges* that lead to *successor* nodes, and *in-edges* that come from *predecessor* nodes. The set $pred[n]$ is all the predecessors of node n, and $succ[n]$ is the set of successors.

for each n
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$
repeat
  for each n
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

Figure 2: Computation of liveness by iteration [1].

Although liveness analysis is great way to find out what registers are allowed to use in the snippets, there are some downsides with this approach. First, at high compiler optimization levels, the register allocation algorithms are usually very good. This means that there are seldom any free

registers to find. On lower optimization or at none at all, the task of finding registers is a much easier one. The second problem is that the liveness analysis is only intra-procedural because it is done on CFGs that only contain information on one function in the program. To know what limits this imposes on liveness analysis, it is necessary to know more about SPARC processor's register-convention.

On SPARC, there exists 32 registers, which are grouped in four different classes: *global*, *local*, *in* and *out* (%g0-7, %l0-7, %i0-7, %o0-7). The local registers are supposed to be scratch registers; the in registers are used to send parameters to functions, and out are used to return values from functions. The global registers are a bit different. They are non-windowed as opposed to the rest. The difference between windowed and non-windowed is that windowed registers automatically are saved between function calls, i.e., the local registers that have the same name in different functions are actually not the same registers. The way this is handled is that the local and out registers are a completely new set, while the out of the previous function becomes the in of the new function, and so on for every function. Non-windowed registers on the other hand are the same between functions. This means that, without us knowing it from our liveness analysis, all the registers, except the local, can be used in another function calling the one we are analyzing. Therefore, it is not safe to assume that registers are free just because they seem to be that within our CFG. Most of the time it is OK to make the assumption mentioned above, but one has to remember that it is not always strictly so.

In addition to the set of 32 ordinary registers, there also exists a set of floating-point registers, some registers to hold integer and floating-point condition codes, as well as a number of other miscellaneous registers. At this moment, the liveness analysis does not handle the floating-point registers. This is because they are all global and we cannot really know if they are used in other functions or not. (We do not use these register in our snippets anyhow.)

The condition code registers are on the other hand used in our snippets, and the SAIT therefore must find out if they are alive or not. Since they are used and defined as ordinary registers, with the exception of implicit use and definition by some instructions, they can easily be analyzed as well. All we have to do is to take special care about instructions that branches on a condition code register or, like `cmp`, sets a condition code register. This problem is no longer present in assembler code for the SPARC V9, since the condition code register has to be explicitly named.

3.3 Handling Delay Slots

The SAIT must correctly handle and instrument instructions that are placed in the delay slots of control-flow instructions. A delay slot means that the instruction after the jump or branch instruction is executed before the jump or branch is executed. This is traditionally done to keep the processor pipeline busy. An example of a load in a delay slot is given below.

```
bne     %reg1, .LABEL
ld      [addr], %reg2   !! delay slot instruction
```

There exists three different kinds of situations that arise with regard to delay slots. They are in principle handled in the same way, but extra care has to be taken in two of the cases. This means that some extra checks has to be performed by the parser and that additional instructions has to be added. The three cases are described in more detail below.

Case #1. The trivial case, without any register-dependencies. If the instruction after the branch is just an ordinary instruction, the branch is just written to the basic block as usual. Otherwise, if it is a load or a store that is to be instrumented, the load/store is written first, then the branch, and finally a `nop` (to fill the delay slot) is written to the basic block.

Original code		*Replaced code*
`bne %reg1, .LABEL` *`ld [addr], %reg2`*	$\Longrightarrow$	*`ld [addr], %reg2`* `bne %reg1, .LABEL` `nop`

Case #2. Sometimes it is not possible to just lift out the instruction in the delay slot and place it before the branch. If the load in the delay slot is actually loading a new value into the register used to decide to branch or not, then this simple strategy would alter the execution path of the program. To avoid this, the content of the register is moved to a temporary, free register (that is found by the tool's liveness analysis) and then this temporary register is instead used in the branch instruction. If there is no free register available, a register is spilt to memory and then this register is used. Then afterwards, the original content of the register is read back from memory again. An example on how this can look is given below.

Original code		*Replaced code*
`bne %reg, .LABEL` *`ld [addr], %reg`*	$\Longrightarrow$	`mov %reg, %temp_reg` *`ld [addr], %reg`* `bne %temp_reg, .LABEL` `nop`

Case #3. Another tricky thing to handle with regard to loads/stores in delay slots are annulling delay slots. Here, depending on if the branch is taken or not, the execution in the delay slot is executed or not. In this case, just moving the load/store is not enough, we also have to do some additional code expansion. First, the original branch is replaced with a branch of the same kind, but with a different destination. The new destination is a new label created by the SAIT. At this label, the load or store in the delay slot is instrumented and executed. If the branch is not taken, we reach another new label, but here the load or store is *not* executed. At the end of both the new blocks, a branch is always taken. In the case where the load or store was executed, this is to the actual label indicated by the original branch; otherwise, the branch is to the label directly following the original branch.

Original code		*Replaced code*
`bl,a,pt %icc,.L900000283` *`st %o0,[%g1+%l2]`* `.L77000552:`	$\Longrightarrow$	`bl,a,pt %icc,.LX686` `nop` `ba .LX687` `nop` `.LX686:` *`st %o0,[%g1+%l2]`* `ba .L900000283` `nop` `.LX687:` `.L77000552:`

3.4 Using the Instrumentation Tool

The SAIT is primarily designed for usage with the DSZOOM system.[2] In DSZOOM, there are three different kinds of snippets: one snippet to insert at global integer loads (`IntLoad`), one snippet for global floating-point loads (`FloatLoad`), and one snippet for global stores (`Store`).

[2]The tool is currently used in several other SPARC-related projects.

```
IntLoad
#
!! The snippet is written as SPARC assembler
#
FloatLoad
#
!! The snippet is written as SPARC assembler
#
Store
!! The snippet is written as SPARC assembler
#
```

Figure 3: Snippet-file layout.

The instrumentation tool is invoked as a normal Java program with two arguments: (1) the *.s-file to instrument and (2) a text file with user-written snippets. The layout of the snippet-file is simple. Three keywords are used to separate different snippets: `IntLoad`, `FloatLoad`, and `Store`. To separate different parts of this file, the character "#" is also used. An example of the layout is shown in Figure 3.

To increase the flexibility in snippets, a group of special symbols is available, in addition to the ordinary SPARC-assembler. A description of all these symbols is given in Figure 4. Finally, certain lines of the snippet-code is possible to prefix with the character "*" and a number. Then, depending on how some conditions are met, only those lines with one of the specific numbers will be inserted into the code. This is an easy way to have a little bit more "intelligent" snippets. An example of a short snippet using this technique is shown below.

```
FloatLoad
#
*1 fcmps $F, $3, $3
*2 fcmpd $F, $3, $3
fbne,pn $F, .LY$L
...
```

In this snippet, depending on if the instrumented instruction it is a single or a double floating-point load, either a compare single (`fcmps`) or a compare double (`fcmpd`) will be inserted by the tool.

4 Low-Level Optimization Techniques

Besides implementing the SAIT for use with the DSZOOM system, a number of other changes has been done to the original system. Among those are some new optimizations to the snippets as well as optimizations to the DSZOOM runtime-system.

4.1 Rewriting Snippets and Reducing the MTAG Size

We have noticed that when the programs are compiled with the highest optimization levels, the liveness analysis seldom founds enough free registers for "old" snippets written for the instrumentation system based on the EEL library. Since EEL could not be used on optimized code, this problem was not identified earlier. Spilling the registers to memory in more than 50% of cases of instrumentation insertion-points is a high price to pay.

$\$n$:	Where n can be 1, 2, or 3. These symbols represent the three arguments of the instrumented instruction. For example, the instruction `ld [%g1+128],%g5`, will give $1=%g1, $2=128, and $3=%g5.
$I:	This represents the instruction instrumented itself. For the same example as above, the $I is equal to: "`ld [%g1+128],%g5`"
$L:	This symbol is replaced with an incremented digit, representing new labels inserted by the parser.
$R:	This symbol either returns a free register found by the liveness analysis or spills (i.e., stores) a register to memory, and thereby makes it available to use in the snippet. The register is restored after the snippet, if any spilling occurred, to maintain the execution-correctness of the instrumented file.
$F:	The same as the $R above, with the exception that the registers looked for are the floating-point condition code registers; that is %fcc0, %fcc1, %fcc2, or %fcc3.
$D:	This symbol is replaced with the type of load that is instrumented, much like the $I, but without any arguments.
$S:	The same as the $D above, but for stores instead.

Figure 4: A list of all special snippet-file characters available.

To solve this problem, we reserve SPARC's application registers (see section 2.2) during the compilation phase of the application. Thus, the SAIT heavily depends on the application register reserved by the compiler, that is registers %g2, %g3, and %g4 for our target architecture. Therefore, all snippets are rewritten to use only those three registers. In fact, all snippets can be written with only two reserved registers.

To reduce the cache pollution introduced by the DSZOOM system, we reduced the size of the directory entries and MTAGs (see [16] for more details). One byte (instead of two) is now used to store the information for both data structures. This optimization also removes one load instruction from the store snippet.

4.2 Straight Execution Path

It is common that most of the code in the snippet in most cases will never be executed. It only takes up place in the instruction cache. Thus, "taking away" the part of the code that is not frequently used will produce a "straighter" execution path for many of the instrumented applications. Basically, we divide snippets into two parts: a *fast-path* part, and a *slow-path* part (see Figure 5 for an example). This functionality is also implemented in SAIT for all types of instrumented loads/stores.

4.3 Avoiding Local Load/Store Instrumentation

It can be hard at instrumentation time to know what loads and stores are global/shared. If we do not know this, there is a risk that SAIT instruments even some local loads/stores, what leads to a larger instrumentation overhead than necessary. One simple method to identify local loads/stores is to find out if the effective address of a load/store comes from a constant. On SPARC, the

```
FloatLoad
#
!! This is the fast-path part of the snippet
!! The code is placed next to the instrumented instruction
  $I                        !! the original instruction
  *1 fcmps $F, $3, $3
  *2 fcmpd $F, $3, $3
  fbne,pn $F, .LY$L         !! if (NaN), goto slow-path
  add $1, $2, %g3           !! %g3 = effective address
.LQ$L:                      !! label generated by SAIT
#
FloatLoad2
#
!! This is the slow-path part of the snippet
!! The code is placed at the end of the instrumented procedure
.LY$L:                      !! label generated by SAIT
  srl %g3, 28, %g4          !! range check
  sub %g4, 8, %g4
  brnz,pt %g4, .LQ$L        !! if (local_ load) goto LQ-label
  nop
  save %sp, -112, %sp
  mov %y, %l0               !! save %y register
  mov %g1, %l1              !! save %g1 register
  mov %ccr, %l2             !! save %ccr register
  mov %fprs, %l3            !! save %fprs register
  mov %g5, %l5              !! save %g5 register
  call DSZOOM_load_coherence_routine
  mov %g3, %l7              !! save the original effective address
  $D [$l7], $3              !! original load
  stb $g4, [$g3]            !! release and update dir_ entry
  mov %l0, %y               !! restore %y register
  mov %l1, %g1              !! restore %g1 register
  mov %l2, %ccr             !! restore %ccr register
  mov %l3, %fprs            !! restore %fprs register
  mov %l5, %g5              !! restore %g5 register
  restore
  ba .LQ$L                  !! goto end of the fast-path part
  nop
#
```

Figure 5: The two-part snippet example.

constants are constructed via the use of the instruction `sethi`. This instruction sets the highest 22 bits of a register to a constant. Then, using the assigned register and another constant, expressed via `lo(some_name)`, a load or store from a constant address can be done. The look of such an instruction is, e.g., `ld [%g1+lo(num_rows)],%g5`. Therefore, choosing not to instrument loads/stores where the argument includes a register plus a `lo`-construct, leads to avoiding some of the local memory accesses otherwise instrumented. This simple strategy is implemented in the instrumentation tool. In addition, explicit accesses to the stack are ignored by SAIT. The explicit stack-references are built up from two SPARC-specific registers: %sp or %fp.

4.4 Write Permission Cache (WPC)

The code for a store snippet is substantially more complicated compared with load snippets, as shown by this pseudo code snippet:

Original instruction		*Replaced by code snippet*
`ST Rx, addr`	$\Longrightarrow$	`LOCK (MTAG_lock[addr]);` `LD R7, MTAG_value[addr];` `if (R7 != 1)` `  call StoreProtocol;` `ST Rx, addr;` `UNLOCK (MTAG_lock[addr]);`

Much of the complication comes from the fact that we allow for more than one thread to run in each node. Before a store can be performed, the cache line's write permission in the local structure "MTAG" should be consulted. However, in order to avoid corner cases where the cache lines gets downgraded between the MTAG consultation and the point in time where the store is performed, the MTAG entry must be locked before the consultation. The content of the MTAG may reveal the need to call the store coherence routine (`StoreProtocol` in the code above) in order to get write permission, after that the store can be performed and the MTAG lock released. Note the use of a dedicated register R7 in the example above. The R7 register has to be a globally reserved register, or an unused register found by SAIT's liveness analysis.

The store snippet currently is responsible for about half of the instrumentation overhead. Reducing the load instrumentation would expose the efficiency of store instrumentation further. To cut the store snippet overhead, we introduce a *write permission cache* (WPC). The idea is the following: when a thread has ensured that it has the write permission for a cache line, it holds that permission hoping that following stores will be to the same cache line (spatial locality). The address/ID of the cache line is stored in a dedicated register (in our implementation: %g2). If indeed the next store is to the same cache line, the store snippet is reduced to conditional branch operation, i.e., no extra memory instructions need to be added, as shown by this pseudo code:

Original instr.		*Fast-path snippet:*	*Slow-path*
`ST Rx, addr`	$\Longrightarrow$	`if (Rwpc != addr)` `  call Slow-path;` `ST Rx, addr;`	`UNLOCK (MTAG_lock[Rwpc]);` `LD Rwpc, #addr;` `LOCK (MTAG_lock[addr]);` `LD R7, MTAG_value[addr];` `if (R7 != 1)` `  call StoreProtocol;`

`Rwpc` denotes a reserved register used to store the address held in the WPC. It should be noted that the fast-path of this store snippet only consists of ALU instructions for "hits" in the write permission cache. One possible UltraSPARC implementation of a 1-entry WPC is shown in Figure 6. The code is for 64 bytes cache lines and 1-byte MTAG entries, 32-bit address space.

```
Store
#
!! This is the fast-path part of the snippet
  add $2, $3, %g4          !! %g4 = effective address
  srl %g4, 6, %g3          !! %g3 = cache line ID
  sub %g3, %g2, %g3        !! %g2 = Rwpc
  brnz,pt %g3, .LY$L_WPC_MISS
  nop
.LQ$L_MSTATE:              !! label generated by SAIT
  $I                       !! the original instruction
#
Store2
#
!! This is the slow-path part of the snippet
.LY$L_WPC_MISS:            !! label generated by SAIT
  srl %g2, 22, %g3         !! range check
  sub %g3, 8, %g3
  brnz %g3, .LQ$L_FALSE_MTAG
  nop

  sethi %hi(0x8dc00000), %g4
  add %g4, %g2, %g4
  stb %g0, [$g4]           !! release and update old MTAG

.LQ$L_FALSE_MTAG:          !! label generated by SAIT
  add $2, $3, %g4          !! %g4 = effective address
  srl %g4, 28, %g3         !! range check
  sub %g3, 8, %g3
  brnz,pt %g3, .LY$L_MSTATE
  srl %g4, 6, %g2          !! %g2 = new Rwpc

  sethi %hi(0x8dc00000), %g4
  add %g4, %g2, %g4        !! %g4 = new MTAG addr

.LQ$L_PREV:                !! label generated by SAIT
  ldstub [$g4], %g2        !! lock new MTAG
  sub %g2, 255, %g3
  brz,pn %g3, .LQ$L_PREV
  nop

  add $2, $3, %g3          !! %g3 = effective address
  brz,pn %g2, .LY$L_MSTATE  !! check if in the M-state
  srl %g3, 6, %g2          !! %g2 = new Rwpc

  ... call DSZOOM_store_coherence_routine ...

  ba .LQ$L_MSTATE
  nop
#
```

Figure 6: The UltraSPARC implementation example of a 1-entry WPC.

5 Performance Study

5.1 Experimental Setup

All the experiments are performed on the same system that was used to test and develop the original implementation of DSZOOM (using EEL). In this paper, we use Sun's Forte Developer 6.2 C Patch 111679-08 compiler instead of the GNU's gcc-2.8.1 compiler. For hardware details, see the original DSZOOM paper [16].

5.2 Applications

To test the performance of this new DSZOOM system, we use the well-known scientific workloads from the SPLASH-2 benchmark suite [29]. The sizes of the data-set used and the un-instrumented uniprocessor execution times are presented in Table 1. The programs are compiled both without optimizations, using the cc's `-xO0` flag, and with the highest optimization levels, with the `-fast` flag. The demonstrated speedup for optimized applications is over four times (average). In this setup, we are able to execute all applications without any modifications, except for Volrend. The reason why we cannot run Volrend is that the global variables are used as shared. It should be possible to modify this application to get rid of this problem. We began all measurements at the start of the parallel phase to avoid DSZOOM's run-time system initialization.

Program	**Problem Size**	**Seq. Time -xO0 [sec.]**	**Seq. Time -fast [sec.]**
FFT	1,048,576 points (48.1 MB)	14.29	3.18
LU-c	1024×1024, block 16 (8.0 MB)	66.61	13.56
LU-nc	1024×1024, block 16 (8.0 MB)	80.30	30.56
Radix	4,194,304 items (36.5 MB)	30.95	6.67
Barnes	16,384 bodies (32.8 MB)	57.02	13.28
Cholesky	tk29.0 (25.3 MB)	20.18	3.45
FMM	32,768 particles (8.1 MB)	117.58	25.03
Ocean-c	514×514 (57.5 MB)	46.76	14.61
Ocean-nc	258×258 (22.9 MB)	18.32	3.72
Radiosity	room (29.4 MB)	28.98	11.10
Raytrace	car (50.2 MB)	11.28	3.89
Water-nsq	2197 molecules, 2 steps (2.0 MB)	134.47	26.07
Water-sp	2197 molecules, 2 steps (1.5 MB)	34.33	7.76

Table 1: Data-set sizes and sequential-execution times for non-instrumented SPLASH-2 applications, compiled with `-xO0` (no optimizations) and `-fast` (high compiler optimization) flags.

5.3 Performance Overview

There are some initial overheads for our new instrumentation system. To avoid register spilling during the instrumentation phase, all applications are compiled with the `-xregs=no%appl` flag. That reserves three application registers that the compiler otherwise could have used to optimize the compiled code even further (typically to reduce the number of loads/stores in the application). The SAIT is also sometimes replacing loads and stores from delay slots with our code snippets, what "work against" already performed compiler optimizations. Table 2 shows the results of those two initial slow-downs. The exclusion of the application registers does not really affect the

Program	With Appl. Regs	Without Appl. Regs	Overhead	"Empty" Delay Slots	Overhead
FFT	3.36	3.37	1.00	3.18	0.95
LU-c	13.15	13.27	1.01	13.62	1.04
LU-nc	29.71	30.16	1.02	30.61	1.03
Radix	6.96	6.96	1.00	6.66	0.96
Barnes	12.99	13.05	1.00	13.24	1.02
Cholesky	3.39	3.40	1.00	3.47	1.02
FMM	24.10	22.99	0.95	28.35	1.18
Ocean-c	15.19	15.17	1.00	14.66	0.97
Ocean-nc	3.83	3.86	1.01	3.73	0.97
Radiosity	11.27	11.04	0.98	11.03	0.98
Raytrace	3.74	3.76	1.01	3.97	1.06
Water-nsq	24.22	24.52	1.01	26.35	1.09
Water-sp	7.30	7.36	1.01	8.01	1.10
Average	**12.25**	**12.22**	**1.000063**	**12.84**	**1.03**

Table 2: The original overhead built into this new system. The time is given in seconds.

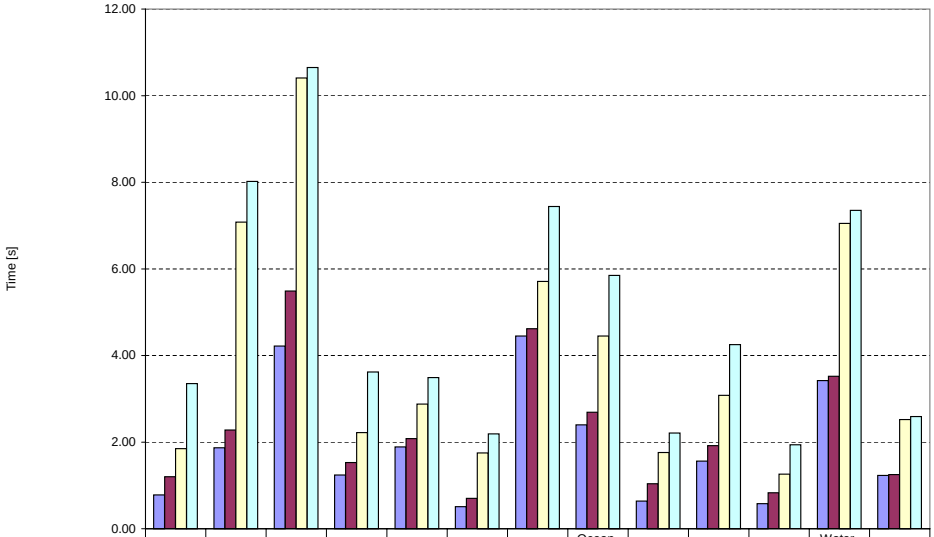
execution time, but moving the loads and stores from the delay slots slows down the programs with about 3%.

Sequential performance after instrumentation. Table 3 shows the performance for instrumented sequential programs for two different snippets. The first snippet is just the "normal," somewhat improved snippet that was also used in the original DSZOOM system (see section 4.1). The second snippet uses the technique of dividing the snippet in two parts, thereby getting a straighter execution path in most cases (this is described in section 4.2). The "straight code" optimization is positive for all applications, except for Water-nsq. The percentage of statically replaced loads and stores is also shown in this table. Those numbers are quite high and result in a rather large number of local loads/stores that are instrumented for some programs.

By profiling the DSZOOM system, we have noticed that there are many local/non-shared loads and stores that are still instrumented by SAIT even though the SAIT is trying to avoid many stack/static references (see section 4.3 for more details). To avoid local load/store-instrumentation, we perform a two-phase instrumentation. This instrumentation is based on the profile data generated from the first run of the instrumented application. In the second instrumentation phase, we simply avoid to instrument loads/stores that were classified as local from the first run. The results obtained using this technique (we call it for *program slicing* in this paper [28]) are given in Table 4. The number of statically replaced loads and stores is drastically reduced compared to the numbers from Table 3. These numbers are relevant for the instrumentation system that could be integrated with the higher levels of a compiler, such as the OpenMP compiler [4], the Unified Parallel C (UPC) compiler [2, 5], or the JIT code generator of a Java system [6]. In the same table, we also present sequential overheads for a 1-entry WPC implementation from Figure 6 (the `DSZOOM_store_coherence_routine` is never called in this case since the sequential protocol is always in the *M-state*). Program slicing is an effective method for Barnes, Radiosity, Raytrace, and Water applications. The WPC implementation, on the other hand, can lower the instrumentation overheads for the following applications: FFT, LU-c, and LU-nc.

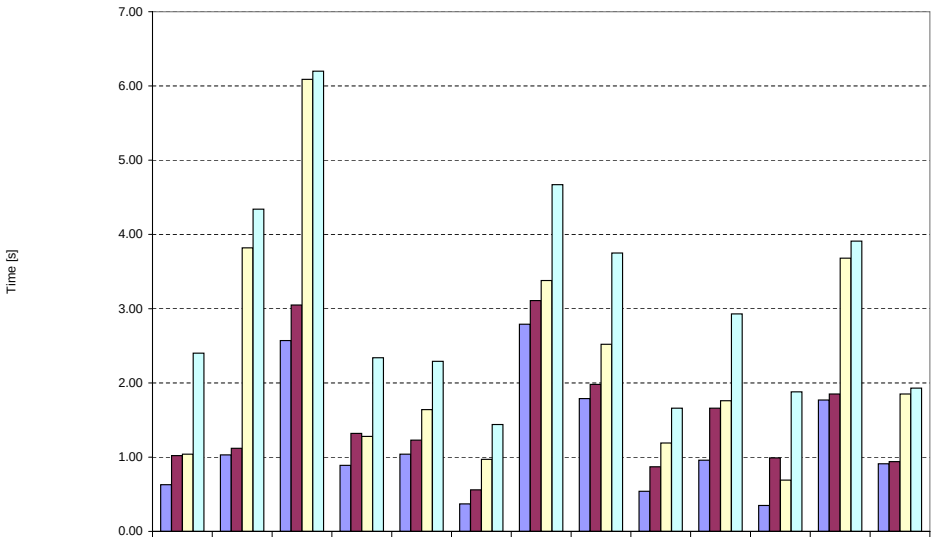
Parallel performance. Figure 7 shows the execution times in seconds for several different configurations for 8- and 16-processor runs. All applications are compiled with the `-fast` flag and instrumented with SAIT with two-part snippets (section 4.2). Figure 8 shows the execution times in seconds for *sliced* programs (applications with only global/shared memory access instrumentation) and *sliced* & 1-entry WPC implementation.

8 processors



	FFT	LU-c	LU-nc	Radix	Barnes	Cholesky	FMM	Ocean-cont	Ocean-nc	Radiosity	Raytrace	Water-nsq	Water-sp
E6000 8 CPUs	0.78	1.87	4.22	1.24	1.89	0.51	4.45	2.40	0.64	1.56	0.58	3.42	1.23
CC-NUMA 2x4	1.20	2.28	5.49	1.53	2.08	0.70	4.62	2.69	1.04	1.92	0.83	3.52	1.25
DSZOOM-WF 1x8	1.85	7.08	10.41	2.22	2.88	1.75	5.71	4.45	1.76	3.08	1.26	7.05	2.52
DSZOOM-WF 2x4	3.35	8.02	10.65	3.62	3.49	2.19	7.44	5.85	2.21	4.25	1.94	7.35	2.59

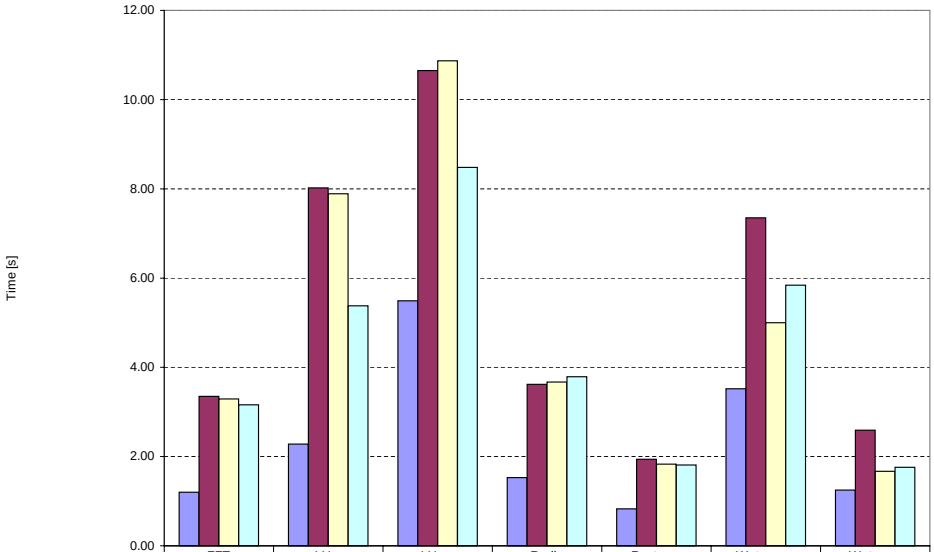
16 processors



	FFT	LU-c	LU-nc	Radix	Barnes	Cholesky	FMM	Ocean-cont	Ocean-nc	Radiosity	Raytrace	Water-nsq	Water-sp
E6000 16 CPUs	0.63	1.03	2.57	0.89	1.04	0.37	2.79	1.79	0.54	0.96	0.35	1.77	0.91
CC-NUMA 2x8	1.02	1.12	3.05	1.32	1.23	0.56	3.11	1.98	0.87	1.66	0.99	1.85	0.94
DSZOOM-WF 1x16	1.04	3.82	6.09	1.28	1.64	0.97	3.38	2.52	1.19	1.76	0.69	3.68	1.85
DSZOOM-WF 2x8	2.40	4.34	6.20	2.34	2.29	1.44	4.67	3.75	1.66	2.93	1.88	3.91	1.93

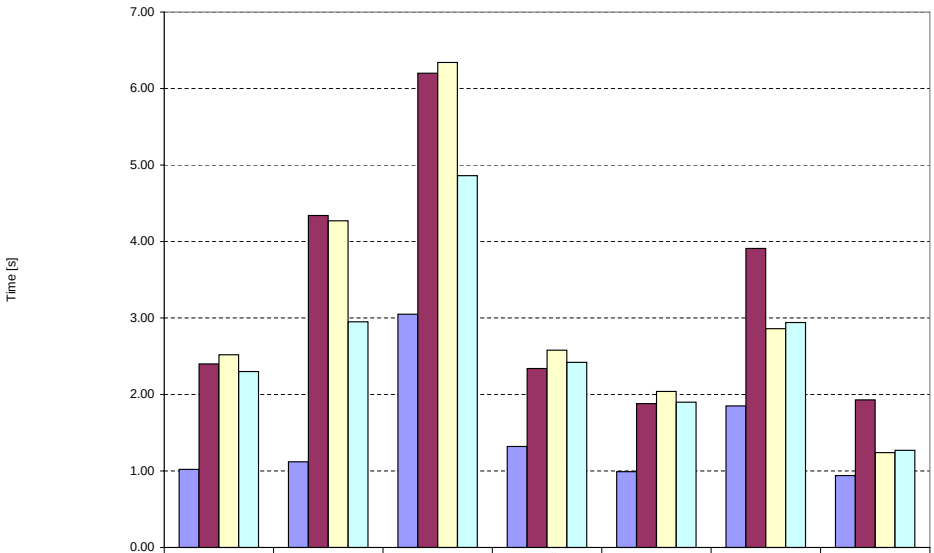
Figure 7: Execution times in seconds for 8- and 16-processor runs for Sun Enterprise E6000, 2-node Sun WildFire (CC-NUMA), single-node DSZOOM-WF, and a 2-node DSZOOM-WF.

8 processors



	FFT	LU-c	LU-nc	Radix	Raytrace	Water-nsq	Water-sp
CC-NUMA 2x4	1.20	2.28	5.49	1.53	0.83	3.52	1.25
DSZOOM-WF 2x4 (a)	3.35	8.02	10.65	3.62	1.94	7.35	2.59
DSZOOM-WF 2x4 (b)	3.29	7.89	10.87	3.67	1.83	5.00	1.67
DSZOOM-WF 2x4 (c)	3.16	5.38	8.48	3.79	1.81	5.84	1.76

16 processors



	FFT	LU-c	LU-nc	Radix	Raytrace	Water-nsq	Water-sp
CC-NUMA 2x8	1.02	1.12	3.05	1.32	0.99	1.85	0.94
DSZOOM-WF 2x8 (a)	2.40	4.34	6.20	2.34	1.88	3.91	1.93
DSZOOM-WF 2x8 (b)	2.52	4.27	6.34	2.58	2.04	2.86	1.24
DSZOOM-WF 2x8 (c)	2.30	2.95	4.86	2.42	1.90	2.94	1.27

Figure 8: Parallel performance for 8- and 16-processor runs. Execution times in seconds for a 2-node Sun WildFire (CC-NUMA), (a) 2-node DSZOOM-WF (from Figure 7), (b) 2-node DSZOOM-WF with only global/shared memory accesses, and (c) a 2-node DSZOOM-WF with only global/shared memory accesses and a 1-entry WPC.

Program	% Loads Replaced	% Stores Replaced	Orig. Snippet (Section 4.1)	Straight Code (Section 4.2)	Relative Speedup
FFT	50.2%	45.1%	10.19	9.93	0.97
LU-c	40.6%	33.2%	51.74	51.19	0.99
LU-nc	43.0%	36.1%	68.18	66.80	0.98
Radix	32.3%	29.1%	10.95	10.64	0.97
Barnes	59.1%	64.4%	21.14	20.13	0.95
Cholesky	60.5%	43.8%	12.33	11.83	0.96
FMM	64.8%	51.7%	42.17	37.45	0.89
Ocean-c	63.5%	69.9%	30.91	N/A	N/A
Ocean-nc	41.7%	65.6%	8.03	N/A	N/A
Radiosity	69.8%	64.0%	23.32	21.99	0.94
Raytrace	63.5%	55.6%	10.60	9.01	0.85
Water-nsq	42.6%	32.6%	51.14	52.02	1.02
Water-sp	37.2%	27.5%	16.09	14.96	0.93
Average	**51.4%**	**47.6%**			**0.95**

Table 3: Sequential execution times in seconds for instrumented SPLASH-2 applications, compiled with the cc's `-fast` flag. The percentage of statically replaced loads and stores is given in the second and the third column, respectively.

Finally, in Figure 9, we compare the performance of a DSZOOM system that uses SAIT and the new optimizations available, and an earlier DSZOOM system that uses EEL [16].

5.4 WPC Study

The write permission cache optimization is introduced in section 4.4. In this section we perform several experiments (on sequential code) with different WPC settings to see how much the instrumentation overhead could be reduced if there is some spatial locality for memory-store operations. We report the average number of consecutive stores to the same cache line (before the MTAG should be released) for different cache line settings (64, 128, and 256 bytes) and for a 1- and 2-entry WPC. Results are shown in Figure 10. There is a large amount of spatial locality for stores. The WPC with two entries of 256 bytes each could reduce the sequential instrumentation overhead by several factors for especially LU-c, Cholesky, and Ocean-c.

6 Conclusions and Future Work

In this paper, we have described SAIT, a SPARC assembler instrumentation tool. SAIT is today a crucial part of the DSZOOM instrumentation system. It can instrument programs compiled with modern compilers and with the highest compiler optimization levels. The instrumentation overhead for sequential execution of the SPLASH-2 benchmarks instrumented with DSZOOM-related snippets ranges from around 30% for the un-optimized programs to around 100% for programs with high optimization. Still, the actual execution time is lower for the optimized programs than the un-optimized ones. The performance improvements range from 1.07 to 2.82 times (average 1.73) if we compare this new DSZOOM instrumentation system with the old one from the original DSZOOM paper. Writing and changing snippets with SAIT is an easy task.

Currently, SAIT is not performing any optimizations after the instrumentation phase. We would like to extend our tool to perform code scheduling and register allocation after the instrumentation. Alternatively, the instrumentation could be integrated with the higher levels of a compiler, such

`-fast` compiler flag (optimized code)

Program	% Loads Replaced	% Stores Replaced	Original Overhead	Program Slicing	Slicing & 1-entry WPC
FFT	15.1%	17.8%	212.9%	206.3%	164.2%
LU-c	13.5%	16.0%	276.8%	279.3%	147.1%
LU-nc	13.1%	15.1%	119.8%	122.3%	65.2%
Radix	8.8%	16.6%	59.5%	58.9%	75.4%
Barnes	19.8%	21.4%	50.8%	15.1%	19.0%
Cholesky	16.8%	13.7%	242.6%	243.2%	345.2%
FMM	N/A	N/A	53.3%	N/A	N/A
Ocean-c	39.7%	52.5%	115.1%	116.1%	90.9%
Ocean-nc	31.9%	50.7%	117.5%	114.5%	106.2%
Radiosity	N/A	N/A	98.8%	11.7%	15.7%
Raytrace	20.8%	14.8%	131.9%	68.9%	67.1%
Water-nsq	17.1%	15.4%	100.8%	27.7%	21.6%
Water-sp	19.0%	12.0%	90.9%	20.7%	18.6%
Average	**19.6%**	**22.4%**	**128.5%**	**107.1%**	**94.7%**

`-xOO` compiler flag (un-optimized code)

Program	% Loads Replaced	% Stores Replaced	Original Overhead	Program Slicing	Slicing & 1-entry WPC
FFT	N/A	N/A	87.8%	90.1%	43.3%
LU-c	N/A	N/A	123.9%	122.5%	41.1%
LU-nc	N/A	N/A	103.8%	103.1%	36.0%
Radix	N/A	N/A	17.8%	12.5%	19.6%
Barnes	N/A	N/A	28.2%	1.3%	0.7%
Cholesky	N/A	N/A	72.2%	81.2%	65.7%
FMM	N/A	N/A	22.6%	9.3%	8.5%
Ocean-c	N/A	N/A	72.3%	66.7%	36.0%
Ocean-nc	N/A	N/A	51.3%	42.6%	27.1%
Radiosity	N/A	N/A	40.1%	16.5%	16.2%
Raytrace	N/A	N/A	130.7%	5.3%	5.7%
Water-nsq	N/A	N/A	111.6%	39.3%	39.7%
Water-sp	N/A	N/A	116.1%	41.9%	39.6%
Average	N/A	N/A	**75.3%**	**48.6%**	**29.2%**

Table 4: Sequential performance overhead for "original snippets" from Table 3, for instrumentation of only global/shared memory accesses with a program slicing technique, and in the last column, the overhead for program slicing with a 1-entry WPC (from Figure 6). The "original overhead" is calculated with values from Table 1.

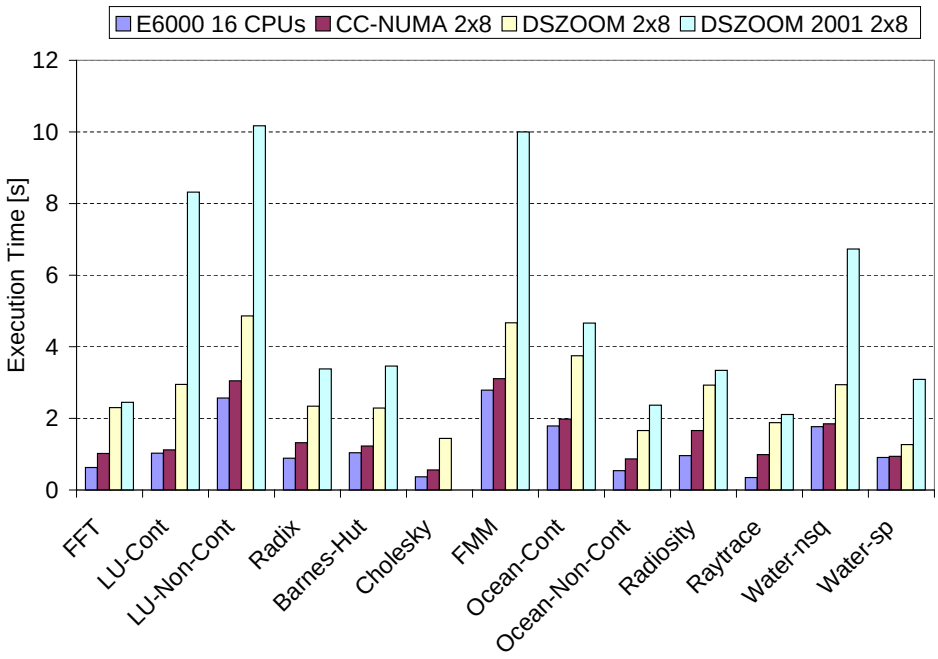
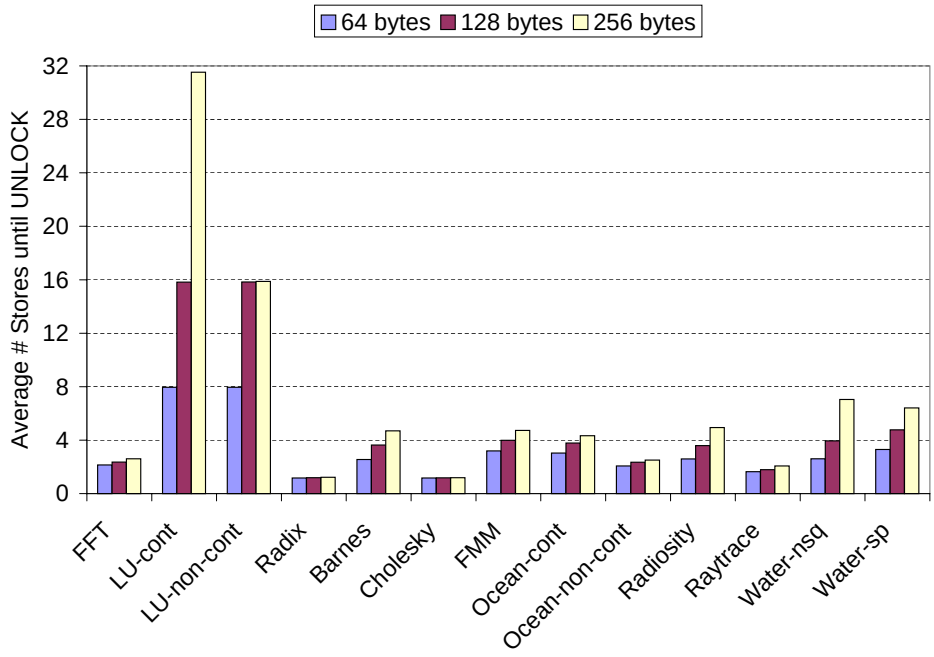


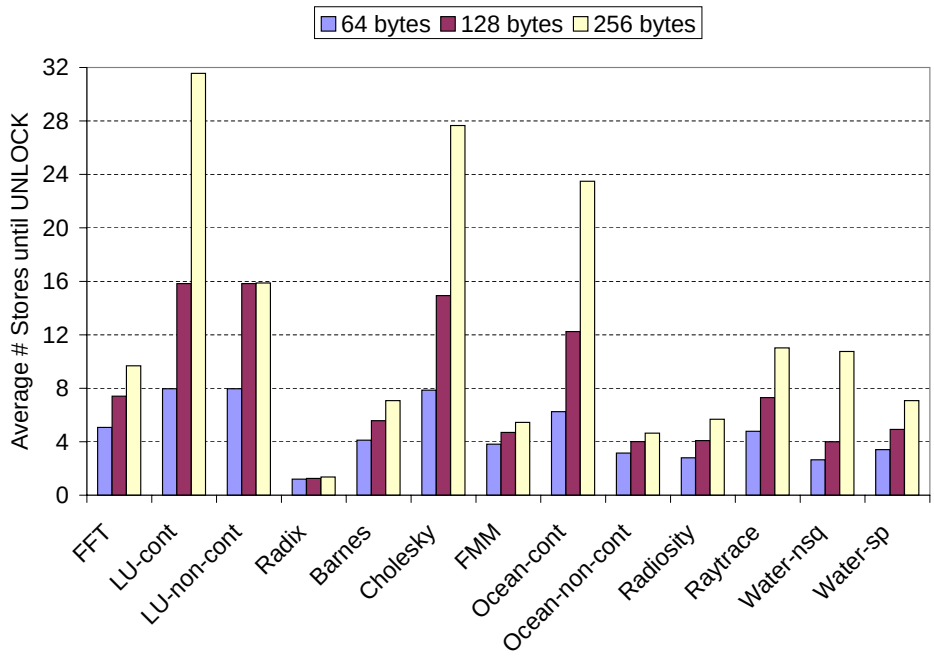
Figure 9: A comparison between Sun Enterprise E6000, a 2-node Sun WildFire, a 2-node DSZOOM-WF compiled with the `-fast` flag and instrumented with SAIT, and a 2-node DSZOOM-WF from 2001 [16].

as the GNU's gcc compiler, the OpenMP compiler, or the JIT code generator of a Java system, and rely on their optimization phases.

We also introduce a write permission cache optimization in section 4.4. We have performed some initial experiments and shown that there is a large amount of spatial locality for stores and that an WPC with two entries of 256 bytes each would reduce the instrumentation overhead by several factors. However, using a WPC also raises some correctness, liveness, and performance concerns. The WPC entries have to be released at synchronization points, at failure to acquire a directory entry or MTAG, and at thread termination. This allowed us to correctly and efficiently run all the applications in the SPLASH-2 benchmark suite. However, this is clearly not sufficient for cases that are more general. More attention should be added to this matter.



(a) 1-entry WPC



(b) 2-entry WPC

Figure 10: The average number of consecutive stores to the same cache line for different cache line sizes, using a WPC with (a) one entry and (b) two entries, respectively.

References

[1] A. W. Appel. *Modern Compiler Implementation in ML.* The Press Syndicate of the University of Cambridge, 1998.

[2] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, The George Washington University, May 1999.

[3] C. Cifuentes, M. Van Emerik, N. Ramsey, and B. Lewis. *The University of Queensland Binary Translator (UQBT) framework*, December 2001. `http://www.itee.uq.edu.au/cms/uqbt.html`.

[4] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan.-Mar. 1998.

[5] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *Proceedings of Supercomputing 2002*, Baltimore, Maryland, USA, November 2002.

[6] J. Gosling, B. Joy, and G. Steele. *The Java Programming Language.* The Java Series. Addison-Wesley, 3rd edition, 2000.

[7] O. Grenholm. Simple and Efficient Instrumentation for the DSZOOM System. Master's thesis, School of Engineering, Uppsala University, Sweden, December 2002. UPTEC F-02-096, ISSN 1401-5757.

[8] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, February 1999.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 3rd edition, 2003.

[10] InfiniBand(SM) Trade Association, InfiniBand Architecture Specification, Release 1.0, October 2000. Available from: http://www.infinibandta.org.

[11] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[12] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[13] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[14] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[15] L. Noordergraaf and R. van der Pas. Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of Supercomputing '99*, November 1999.

[16] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of Supercomputing 2001*, Denver, Colorado, USA, November 2001.

[17] D. J. Scales and K. Gharachorloo. Towards Transparent and Efficient Software Distributed Shared Memory. In *Proceedings of the 16th ACM Symposium on Operating System Principles, Saint-Malo, France*, October 1997.

[18] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. Technical Report 97/3, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[19] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–185, October 1996.

[20] I. Schoinas, B. Falsafi, M. Hill, J. R. Larus, and D. A. Wood. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Proceedings of the 6th International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[21] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 297–306, October 1994.

[22] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. In *Proceedings of IEEE Hot Interconnects IV*, pages 41–52, August 1996.

[23] E. Speight and J. Bennett. Brazos: A Third Generation DSM System. In *Proceedings of the 1st USENIX Windows NT Symposium*, August 1997.

[24] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating System Principle*, October 1997.

[25] Sun Microsystems. *ABI Compliance and Global Register Usage in SPARC V8 and V9 Architecture*. `http://soldc.sun.com`.

[26] Sun Microsystems. *C User's Guide Forte Developer 6 update 2*, 2001.

[27] WebGain. *Java Compiler Compiler Grammar File Documentation*, June 2002. `http://www.webgain.com`.

[28] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.