UPTEC F 02 096

Simple and Efficient Instrumentation for the DSZOOM System

OSKAR GRENHOLM

Supervisor: Zoran Radovic

Examiner: Professor Erik Hagersten

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA UNIVERSITY

Simple and Efficient Instrumentation for the DSZOOM System

BY
OSKAR GRENHOLM

December 2002

COMPUTER SYSTEMS
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Master of Science in Engineering
at Uppsala University 2002

Simple and Efficient Instrumentation for the DSZOOM System

Oskar Grenholm
osgr8555@student.uu.se

Computer Systems
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1401-5757
Printed by the Department of Information Technology, Uppsala University, 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 SW-DSM originally developed using EEL (Executable Editing Library) - a binary modification tool from University of Wisconsin-Madison. In this thesis, we identify several weaknesses of this original approach and present a new and simple tool for assembler instrumentation. This tool can instrument (modify) highly optimized compiler output for the newest UltraSPARC processors. Currently, the focus of the tool is load-, store-, and load-store-instrumentation.

We also present several low-level optimization techniques that significantly improve the performance of the DSZOOM system. One of the presented techniques is a store-buffer register optimization, a latency-hiding mechanism for memory-store operations, that can lower instrumentation overheads for some applications (as much as 45% for LU-cont, running on two nodes with 8 processors each).

We also show that by using this new DSZOOM system we execute faster than the old one on all applications in the SPLASH-2 benchmark suite. Improvements range from 1.07 to 2.82 times (average 1.73).

Contents

1 Introduction

Today clusters of symmetric multiprocessors (SMPs) are providing a powerful platform for executing parallel applications. To allow for shared-memory applications to run on such clusters, software distributed shared memory (SW-DSM) systems can help support the illusion of shared memory across the cluster via a software run-time layer between the application and the hardware. This approach can potentially provide a cost-effective alternative to hardware shared memory systems for executing certain classes of workloads. Also the upper scalability of large hardware distributed shared memory systems can extend by connecting several of them via SW-DSM technology.

Most SW-DSM systems keep coherence between page-sized coherence units [Li88], [CBZ91], [KCDZ94]. The normal per-page access privilege of the memory-management unit offers a cheap access control mechanism for these SW-DSM systems. But this large page-size coherence units in the earlier SW-DSM systems created extra false sharing and caused frequent page transfers of large pages between nodes. In order to avoid most of the false sharing, weaker memory models have been used to allow many update actions to be lumped to a specific point in time, such as the lazy release consistency (LRC) protocol [Kel95].

Fine-grain SW-DSM systems with a more traditional cache-line-sized coherence unit have also been implemented. Here, the access control check is either done by altering of the error-correcting codes (ECC) [SFH+96] or by in-line code *snippets* (small fragments of machine code) [SFH+96], [SGT96]. The small cache-line size reduces the false sharing for these systems, but on the other hand the explicit access-control check adds extra latency for each memory access to global data. The most efficient access check reported to date is three extra instructions adding three extra cycles for each load to global data [SFH+98].

At Uppsala University the DSZOOM-WF system has been implemented. This is a sequentially consistent fine-grain distributed software-based shared memory, between the nodes of a Sun-WildFire [HK99] system 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 executable editing library (EEL) [LS95] 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 protocol implemented in C that copies data to the nodes "private" local memory by performing loads and stores from and to remote memory.

This system has been tested and shown to work well with respect to instrumentation overhead. Still this way of instrumenting the binaries directly gives rise to a couple of problems. Mainly this is because of EEL having some limitations in its capabilities to instrument all kind of binaries. Namely EEL does not handle to instrument all cases of instructions placed in delay slots correctly, thus usually making it impossible to instrument any executables compiled with an optimization level above the lowest.

To avoid this limitation we have decided to use an alternative approach to binary instrumentation. Instead we instrument the assembler output from the compiler, and insert the snippets needed.

The compiler finishes it's job of making it all into an executable. By doing the actual instrumentation at the assembler level we can analyze the code and re-arrange the code in a way that let us avoid loads and 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. But this also leads to some limitations of the instrumentation tool, the biggest being that we always have to have the source code of the program we want to instrument. This is not a problem when the EEL is used.

The remainder of this paper is organized as follows. Section 2 gives an introduction to a general DSZOOM system. A presentation of the target architecture used to build DSZOOM on is given in section 3. Section 4 shows how the new Instrumentation tool works. In section 5 we have collected together all the optimizations done to the new system and finally in section 6 we give a overview of the results obtained.

2 DSZOOM Overview

In this section we give a short overview of the DSZOOM system. The DSZOOM proposal is described in a couple of previous papers in more detail by Radovic and Hagersten [RH01a],[RH01b].

Each DSZOOM node could either be a standard single processor machine, a symmetric multiprocessor (SMP), or a CC-NUMA cluster. The node hardware keeps coherence among the caches and the memory within each node. The different cluster nodes run different kernel instances and do not share memory with each other in a hardware-coherent way.

DSZOOM assumes a cluster interconnect with an inexpensive user-level mechanism to access memory located in other nodes, similar to the remote put/get semantics found in the cluster version of the Scalable Coherent Interface (SCI), or the emerging InfiniBand interconnect proposal that supports efficient user-level accesses to remote memory (RDMA READ/WRITE) as well as the atomic operations to smaller pieces of data (CmpSwap and FetchAdd). The remote atomic operation enables the implementation of the *blocking directory protocol*, a hardware coherence protocol implementation presented by Hagersten and Koster [HK99] that eliminates many of the potential race conditions and simplifies the verification of coherence protocols. Most of the complexity of a coherence protocol is related to the race conditions caused by simultaneous requests for the same cache line. In a blocking directory proposal, the processor that has detected the need for global coherence activity will first acquire a lock associated with the cache line before starting the coherence activity itself. The directory entry is locked with either one local atomic memory operation if the entry happens to reside in the same node, or with one remote atomic memory operation if the entry is located in remote node. By implementing the distributed version of the blocking directory protocol entirely in software, DSZOOM has demonstrated that all interrupt- and/or poll-based asynchronous protocol processing, found in almost all traditional SW-DSM implementations, is completely removed by running the entire coherence protocol in the requesting processor. This not only removes the asynchronous overhead, but also makes use of a processor that otherwise would stall.

All SW-DSM systems have to detect accesses to data that is not available locally. In contrast to page-based systems that rely on the virtual memory hardware (page faults) to detect accesses to locally unavailable data (for example, fine-grained SW-DSMs like Shasta [SGT96], [SG97a], [SG97b], Blizzard-S [SFL+94], Sirocco-S [SFH+98], or DSZOOM [RH01b] insert code snippets (small fragments of machine code) to the application binary at loads and stores to perform checks during the runtime if remote accesses are needed or not (we will refer to this operation as an *access control check*). On the upside, the fine-grained approach reduces false-sharing because coherency units are comparable to the hardware implementations, typically 64 bytes large, and accordingly minimizes the need for large unnecessary data transfers. This approach also allows quite efficient implementations of strict memory models, such as sequential consistency (SC). This is very important because many of the popular commercial architectures support only relatively strict memory consistency models, e.g., Intel's x86 architecture supports processor consistency that is a little less strict model compared to the SC. Many of the recent page-based systems use very relaxed memory models such as the lazy release consistency (LRC), in order

to solve false sharing problems that arise from page-sized coherency units. In practice, this will disallow the use of many advanced protocol optimizations for such systems in case of transparent binary execution across the cluster. On the downside, the binary instrumentation technique adds extra latency for each load or store operation to global data, independently if that data is locally available or not. On average, the speedup difference between the original DSZOOM and the hardware-based CC-NUMA system is around 30% for the studied SPLASH-2 applications [WOT+95], where the in-line checks (ILC) for global loads and stores are clearly the largest overhead [RH01b].

DSZOOM uses the executable editing library (EEL) [LS95] to insert fine-grain access control checks after shared-memory loads in a fully compiled and linked executable. This technique is usually called for binary instrumentation. Range checks and node-local MTAG lookups before global stores are also added. All global/shared memory is allocated in the G_MEM area (starting at 0x80000000), that is why DSZOOM must dynamically check if loads and stores are targeting that area or not. Static data and stack accesses are ignored during the instrumentation phase, i.e., they will not be replaced by any snippets. If fine-grain access control check or MTAG lookup fails, the coherence routine is going to be called from DSZOOM's in-line snippets. Global coherence is resolved by a coherence protocol implemented in C that copies data to the node's private local memory by performing loads and stores from remote memory.

3 Target Architecture Overview

In this section some background about the system used to implement DSZOOM on is given. This refers to aspects of both the old and the new version.

3.1 Original Proof-of-Concept Platform

The system used to host the original proof-of-concept DSZOOM was a Sun Enterprise E6000 SMP with 16 UltraSPARC II (250 MHz) processors running Solaris 2.6. The compiler used to compile both EEL and the SPLASH-2 benchmark programs was gcc-2.8.1. The benchmark programs were compiled without any optimization, that is given the flag -O0. This was because EEL could not instrument the binaries produced otherwise. EEL was used to instrument the binaries so that specific code snippets designed for DSZOOM was inserted at loads and stores. The snippets then performed the necessary checks and if necessary called routines implemented in C that kept coherence.

The proof-of-concept implementation was tested thoroughly on this platform and for further information on that system and the results obtained, see [RH01b].

3.2 SPARC V9 ABI Restrictions

In order to make the snippets efficient with respect to register usage, we need to have some free registers at our disposal. The way this has been solved, both in the original DSZOOM and in this new one, is to compile the programs, telling the compiler not to use all of the registers. Thus, those unused registers are free to use as we like.

But for this to work we need to pay attention to be compliant with all SPARC ABI specifications , and especially with global register usage [Sun02]. Currently DSZOOM 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 the 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 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 binary modification process. As mentioned above we need two registers to pass arguments. For this we have choose to use registers %g3 and %g4. On SPARC V8 or V8plus this leaves one extra register, %g2, to use. This register is used to make the snippet a bit more efficient, in such a way that we can take away one or two instructions. There is also another, greater optimization that can be implemented if there is an extra register free. What this is will be mentioned later on in the paper.

As seen, this implementation, only having to use two global registers in the snippet, also allows us to implement DSZOOM on SPARC V9. But then the extra optimizations, made available by the extra application register on SPARC V8 or V8plus, are no longer applicable.

3.3 Compilers

The compiler used to produce the assembler output and the executables is Sun WorkShop 6 update 2 C 5.3 Patch 111679-08. To us there exist a couple of flags available for this compiler that have greater importance than the others. The flags to chose optimization level are among those and the -S flag. A brief overview of these flags, taken from the Forte Developer 6 update 2 manual [Sun01], is given here.

-S Directs cc 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.

float: Allows using the floating-point registers as specified in the SPARC ABI. You can use these registers even if the program contains no floating-point code.

no%float: Does not use the floating-point registers. With this option, a source program cannot contain any floating-point code.

Example: -xregs=appl,no%float. The default is -xregs=appl,float.

-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 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.

4 New DSZOOM Instrumentation

This section contains an argumentation of why this new tool is needed. Then it will explain what is new and how it is implemented.

4.1 Why a New Tool?

As mentioned earlier there are some things with EEL that we see as a problem with regard to what we want to do. The main concern from our side is that there are a great number of binaries that EEL can not instrument. First of all EEL itself is unmaintained. This means that it does not handle to instrument newer binaries, that is programs that are compiled with one of the latest versions of compilers. This can partly be blamed on the fact that EEL is written with a lot of old code (non-ANSI C/C++). Due to this EEL itself can not be compiled using new compilers. Also EEL has a problem to handle hand-written assembler and that does usually lead EEL to make some erroneous analysis in those cases. A third set of binaries that EEL does not always handle correctly is the ones where the .code and .data segments are mixed together. In those cases it is hard knowing what is instructions and what is constants of data. On the other hand this is not a problem when doing the instrumentation at assembler level, where it is always well defined what is code and what is data. Finally, there is the problem that leads to EEL not being able to instrument all binaries compiled with any kind of optimization. Because in binaries that are optimized one of the first optimizations to be applied is the one of putting instructions in the delay slots. Since EEL has a problem instrumenting loads and stores that are placed in delay slots this leads to many optimized binaries not suitable for instrumentation by EEL in the DSZOOM case.

Besides some technical limitations to EEL there is also the issue that it is rather cumbersome to write and change snippets using EEL. What you have to do is to write a program in C/C++ (using EEL) that does the actual instrumentation and if you want to change your snippet you have to rewrite this program and compile it again. Rewriting a snippet is not as simple as you may think. If you want to just add some extra instructions, thereby making the snippet larger, and if you before had some branches you have to change the offset to branch on for all these branches. This is because all branches inserted by EEL into the binary are to an absolute address and not to a label. This is not a problem at all when inserting snippets into assembler code, here you can make up all the new labels you want (as long as they have unique names) and then just branch to them. Overall it is much easier to change and insert snippets using this new instrumentation tool. You just write the snippets in assembler code as text in a text file. Changes can be done in a text editor and then you just invoke the tool to insert these changes into the program to be instrumented.

So, in what ways does this new tool outsmart EEL? First of all it can instrument all kinds of loads and stores in delay slots, thus being able to instrument programs compiled at all optimizations levels. Also it is written to handle assembler output from one of the newest versions of Sun's Forte compiler and therefore handles code written for newer compilers. Another good thing that

comes from instrumenting at the assembler level is the one mentioned above, that you can easily insert new labels in your snippets and thereby allowing for more intelligence in the snippets. Furthermore the idea of inserting only ordinary assembler code and then letting the compiler finish its job, allows for standard binaries, instead of strange looking binaries that do not fall back on any standard, as is the result of EEL's binary instrumentation. This gives that standard tools for examining the binaries, for example the program analyzer, can perform as well as on any other binary.

4.2 The Idea Behind It

The process of making an executable suitable for the DSZOOM system earlier involved the following process: First of all the unmodified SMP application source code written with PARMACS macros is preprocessed with a m4 macro preprocessor. m4 replaces all macros with DSZOOM run-time library calls. Then the the preprocessed file is compiled and linked with the DSZOOM run-time library. The resulting file called the (Un)executable is then passed to the binary modification tool, which is based on an unmodified version of the executable editing library (EEL). The binary modification tool then inserts code snippets, containing fine-grain access control checks after shared-memory loads and range checks node-local MTAG lookups before global stores. The snippets also contain calls to the corresponding coherence protocol routines. This process is shown in Figure 1.

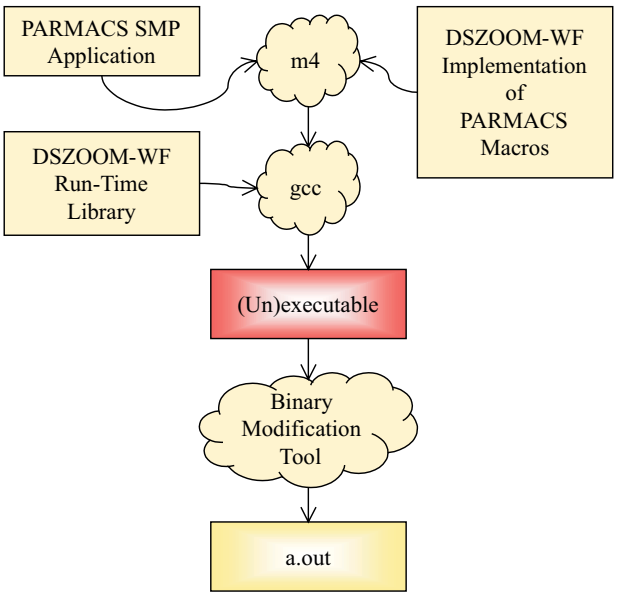


Figure 1: DSZOOM-WF original compilation process.

The problem with this approach comes mainly from limitations in EEL. And the main concern stems from the fact that EEL often has problems instrumenting binaries where the delay slots are

filled with anything else than nops, i.e., empty instructions. Thus if a load or a store is placed in a delay slot, it sometimes can not be instrumented. This limitation makes it impossible for EEL to instrument binaries that have been compiled with any optimization level above the lowest.

Our role in this is to implement a tool that uses a different approach on how to instrument a program and make it into an executable capable of being used in the DSZOOM system. This approach resembles the earlier process a lot, with the difference that the inserting of the code snippets is done on a higher level in the process. Here the assembler output from the compiler is instead analyzed and code snippets are inserted at loads and stores. Then the compiler is called again to finish it's job and make everything into an executable for the DSZOOM system. This executable is now a real executable and nothing further, i.e., binary modification, is needed before it can be run. The changes made to the process is shown in Figure 2.

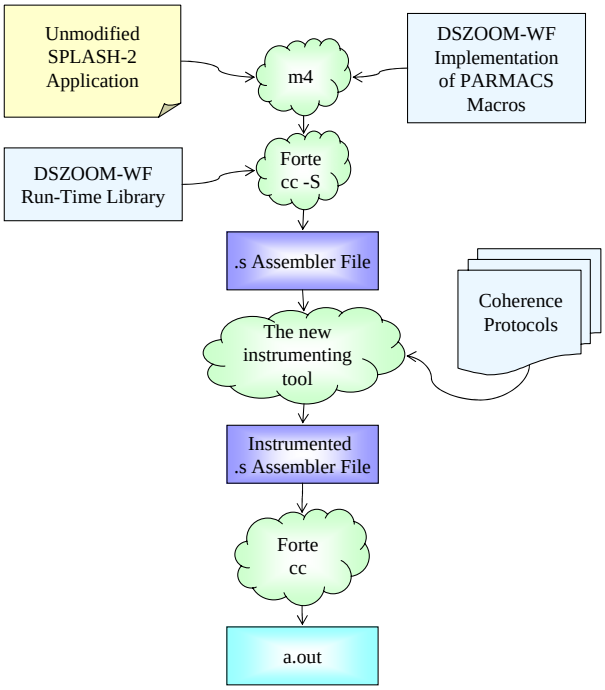


Figure 2: The new compilation process for DSZOOM.

Actually, the changes made are quite small. In practice the same Makefiles can be used with only some minor modifications. An example of how a new Makefile can look is given in Figure 3. The compilation phase is broken in two phases. First the compiler is forced to compile the C

source code and to produce assembler output to a .s file. This is done by giving the compiler the -S flag. Then the tool instruments the .s file and produces a new .s file with all the snippets inserted at the right places. Finally the compiler is used again to compile the .s file and link it with all the necessary libraries to make the executable.

```
.H.h:
	$(M4) $(MACROS) $*.H > $*.h
.C.c:
	$(M4) $(MACROS) $*.C > $*.c
.c.o:
	#This line compiles the .c files into .s files
	$(CC) -S $(CFLAGS) $*.c -o $*.s

	#Here we change directory to the one where the parser is
	cd /parser_dir ; \

	#Here we invoke the parser to instrument the .s files
	#using the snippets given in snippet.txt
	java AsmParser $(DIR_PATH)/$*.s snippet.txt

	#Finally the instrumented .s files are compiled into
	#an executable ready for the DSZOOM system
	$(CC) $(CFLAGS) -c $*.s -lm }
```

Figure 3: An example of how a part of the Makefile for FFT looks. It is using the new Instrumentation Tool.

4.3 Implementation Details

In this section more specific details on how the actual parsing and instrumentation is implemented is given. Also the tools involved in this creation are described.

The instrumentation tool created for use with the DSZOOM system is just an assembler parser with ability to insert code snippets at specified locations. To aid us in the creation of the parser JavaCC [Web02] was used . JavaCC is a parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. Much like the classic 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. The variables can be used to hold special data during the parsing.

4.3.1 Parsing SPARC Assembler

The first thing to do was to try and come up with some kind of grammar for SPARC assembler code. For doing this, and knowing that the most important thing for us was mostly to leave the code as it was and insert extra code at some places, we chose the easiest route possible. Since the code that is supposed to be fed into this parser is generated by a compiler we assumed that

it already had some structure to it. The basic idea for our grammar was to find instructions, see what kind of instruction it was, notice what kind of arguments it had and then store it. Most things that was not an instruction was stored as it was for later, without caring about what it meant. The tokens we used for instructions were of different kinds. One that we used, Instruction3, was a token that recognized generic instructions with three arguments, among others `Add` and `Sub`. And Instruction2 was instructions with two arguments, such as `Mov` and `Cmp`. Other tokens used were Branch, obviously representing branches, for which special care had to be taken, and most important of all, Loads and Stores. Besides those mentioned a couple of other Tokens were used as well. All the Tokens are built up from regular expressions. An example of how the Tokens for Stores and Registers look is given in Figure 4.

The store token:

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

The token representing registers:

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

Figure 4: Two examples of how the tokens in JavaCC looks. They represent different parts of an assembler program and are built up of regular expressions.

The parser goes through the code twice, inserts code snippets at appropriate places and then writes the instrumented code into a file. Between the two passes some additional work is done. Analysis of which registers are used are done (Liveness analysis), and problems regarding the placement of loads and stores in delay slots are analyzed and solved.

First of all this new tool goes through the assembler code and parses it in one sweep. During this pass it writes all of the instructions and arguments into a for this specially created structure. There it is stored for writing back to the instrumented file later. This structure consists of two different classes, implemented in Java, called Basic Block and Control Flow Graph (CFG). The formal definitions of Basic Blocks and Control Flow Graphs [CERL01] can be found in Figure 5. 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 that, until a Branch or a new Label is found, then a new instance of 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 some different kind of ways that a Basic Block usually can end [CERL01]:

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 seen as 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 is seen to belong to the Basic Block as well.) And since we do not try to do inter-procedural analysis we don 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. And finally if the Basic Block ends with a Return statement of some kind, then this indicates that the flow for this function or program ends here and accordingly information about this is stored instead.

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.

Figure 5: The formal definitions of Basic Blocks and Control Flow Graphs.

After the creation of all the Basic Blocks, the flow of those are 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 built 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 straight code or a branch) or a NULL pointer if it is a terminating point.

4.3.2 Liveness

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 statement, `add %g1, %g2, %g3`, registers %g1 and %g2 are used and %g3 is defined. To support this each Instruction writes to its Basic Block which registers it uses and defs. Those are stored in the Basic Block as a hash table. 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, originally implemented in Java, is given here in pseudo-code [App98] in Figure 6. This returns an 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. To know which registers that are free to use just take the once not included in the table.

```
for each n
in[n] <- {}; out[n] <- {};
repeat
    for each n
      in'[n] <- in[n]; out'[n] <- out[n];
      in[n] <- use[n] U (out[n] - def[n]);
      out[n] <- U_{s in succ[n]} in[s];
until in'[n] = in[n] and out'[n] = out[n]
```

Figure 6: Pseudo-code for the algorithm used to analyze the liveness of a function.

Although liveness is a great way to find out what registers you are allowed to use in your snippets, there are some downsides with this implementation. First of all, on higher optimization levels the optimization of the usage of the registers is 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. This is because it is done on CFGs and those only contain information on one function in the program. To know what limits this imposes on liveness it is necessary to know how the register-convention is built up in SPARC.

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 totally 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 both 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 can not really know if they are being used in another function. And besides we do not have any need to know if they are alive or not since we do not use them in our snippets. Registers that we on the other hand use in our snippets and therefore need to do liveness analysis for are the condition code registers. But since they just 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 for the instructions that branches on a condition code register or that, like `Cmp`, sets a condition code register. This problem is no longer present in assembler code for the SPARC V9, since there the condition code register to branch on or set when comparing has to be explicitly named.

4.3.3 Handling Delay Slots

One difference between what this new instrumentation tool can do and EEL, is that it can 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 done to keep the processor pipeline busy. An example of how a load is placed in a delay slot is given here.

```
1:  bne    %reg, .LABEL
2:  ld     [addr], %reg  ! <– 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 way that we have chosen to handle loads and stores in delay slots is quite simple. First all instructions that give rise to delay slots are identified. These are branches of all kinds, calls, jumps and returns. When an ordinary instruction is found what we do is simply, as mentioned earlier, to write that instruction unmodified into a Basic Block. This has to be changed when dealing with delay slots. Here instead when an instruction that has a delay slot after it is found it is not written to the Basic Block, but into a temporary structure. This information is then kept until the next instruction is to be written, then accordingly to what instruction this next one is the information stored is written. Here are two different strategies used. 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.

Case 2

In addition to this there is some extra care that has to be taken in certain cases. 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 contents of the register is moved to a temporary, free register and then this temporary register is instead used in the branch instruction. An example on how this can look is found below.

```
Before:
1:  bne    %reg, .LABEL
2:  ld     [addr], %reg

After:
1:  mov    %reg, %temp_reg
2:  ld     [addr], %reg
3:  bne    %temp_reg, .LABEL
4:  nop
```

If there is no free register available, a register is spillt to memory and then this register is used. Then afterwards, the original content of the register is read back from memory again.

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 of all, 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 our tool. 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 always is 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. The code expansion taken from a real program is in Figure 7.

The effect of all the changes done in this manner to the delay slots in a program has been tested and has been shown to be only a slowdown of about 2-3%.

4.4 Using the Instrumentation Tool

The most important thing this new parser does is to insert the code snippets needed to keep coherence, as implemented by DSZOOM. In DSZOOM, there are three different kinds of snippets that

```
  //Original branch and store
1:    bl,a,pt  %icc,.L900000283
2:    st %o0,[%g1+%l2]          !  <– Store to be instrumented
3: .L77000552:

//After the code expansion
1:    bl,a,pt  %icc,.LX686
2:    nop
3:    ba .LX687
4:    nop
5: .LX686:
6:    //Instrumentation inserted here
7:    st %o0,[%g1+%l2]          !  <– Original store
8:    //and here
9:    ba  .L900000283
10:   nop
11: .LX687:
12: .L77000552:
```

Figure 7: How the annulling branch case is handled.

are needed. These are, one snippet to insert at global integer loads, one snippet for global floating point loads and one snippet for global stores. The instrumentation tool is invoked as a normal java program, giving as arguments the file to instrument and a text file containing the snippets to be inserted. Among the first things the parser do is to read the text file with the snippets, parse and classify them as the proper kind of snippet, i.e., integer load, floating-point load or store. They are now available for the parser to insert at the correct places. The layout of this text file is simple. First is declared what kind of snippet that is described, using the keywords `IntLoad`, `FloatLoad` or `Store`. Then the specified snippet is written. This is then repeated until all snippets are described. To separate the different parts the character # is used. An example of the layout is given below.

```
IntLoad
#
!Here the snippet is given as assembler
#
FloatLoad
#
!Here the snippet is given as assembler
#
Store
!Here the snippet is given as assembler
#
```

To allow for some further flexibility in the snippets, a group of special symbols are available for use in the snippets, in addition to ordinary assembler. A description of all these symbols is in Figure 8.

Finally, another thing that can be done is to prefix certain lines of codes with the character * and a number. Then, depending on how some conditions are met, only those lines with one of

$1, $2, $3: These symbols represent the three arguments of the instruction instrumented. As an example, the instruction: ld [%g1+128], %g5,
will give $1=%g1, $2=128 and $3=%g5.
$I: This represents the instruction instrumented itself. That is, with arguments and all.
$L: This symbol is replaced with an incremented digit, representing new labels inserted by the parser.
$R: This either returns a free register found by the liveness analysis or spills (i.e., stores) a register to memory, and thereby making it available to use in the snippet, and after the snippet, loads the earlier value back into the register from memory.
$F: Here the same thing as with $R is done, with the exception that the registers now looked for are the floating-point condition code registers. That is %fcc0-3.
$D: This symbol is replaced with the type of load that is instrumented, much like the $I, but without the arguments.
$S: The same as $D above, but for stores instead.

Figure 8: A list of all the special characters available.

the specific numbers will be inserted into the code. This is an easy way to have a little bit more intelligent snippets. The conditions on which the choices are made have to be programmed into the tool itself and can not be changed afterwards without recompiling the tool. An example of a short snippet using this technique is written out below. In this snippet, depending on if it is a single or double floating-point load, it will be either a compare single (`fcmps`) or a compare double (`fcmpd`).

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

4.5 Limitations

There are some new features to this new optimization tool compared to EEL. But obviously there are also many areas where we do not perform as well as EEL. The major thing is that while EEL can instrument all types of instructions, our tool can only instrument loads and stores. Also, another limitation that we have, and that EEL does not, is that for our tool to work we need to have the source code of the program we want to instrument. This means that we can not instrument all commercial programs. On the other hand when it comes to scientific programs, the source code is often available. Another thing is that at this stage of the development of the tool, it assumes that the compiler being used is Forte V6.2. For example it can not handle compiler output produced by GNU gcc. (On the other hand implementing support for this would not be a major task.)

Today, there has been no real testing of DSZOOM on a 64-bit system. Just taking DSZOOM, with snippets and everything, and move, would not work right now. This is because that at the moment we do use all the three application registers available in the SPARC V8 or V8plus ABI. But on SPARC V9 (64-bit) there are only two application registers. On the other hand, there is no reason why this should not work if we rewrote the snippets some. Actually, we can manage well with just two application registers. The only downside being that we might have to use some extra instructions in the snippets.

Another small limitation is that you have to change all the makefiles for the programs you would want to instrument. This can easily be avoided if the cc command was overload with the commands needed to perform the instrumentation.

5 Low-Level Optimization Techniques

Besides implementing the new instrumentation tool 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. To show what has been done, first the snippets of the original proof-of-concept system are shown, and then the new snippets with some explanations to what has been changed and why.

5.1 Proof-of-Concept Snippets

Here all the snippets used within the original DSZOOM system are shown. They have been rewritten in the syntax of the new instrumentation tool.

5.1.1 Integer Load Snippets

The original snippet for integer loads is given here, written in the syntax of this new instrumentation tool. The purpose of this snippet is to first check whether or not the value loaded is correct, then if not, to check if it comes a global load and if it is, to call a C routine to handle the coherence.

```
!ORIG_LOAD: ld [%o7 + 892], %o0

  mov           %o7, $R
  add           %o7, 892, %l2     ! Obs!! if %l2 == %o7
  ld            [%l2], %o0
  add           %o0, 1, %g3
  srl           %g3, 0, %g3
  brnz,pn       %g3, .L1
  nop
  srl           %l2, 29, %g3
  sub           %g3, 4, %g3
  brnz,pn       %g3, .L1
  nop
  call          DSZOOM_mem_load   ! %o7 changes
  mov           %l2, %g3          ! delay slot
  ld            [%l2], %o0
  sth           %g4, [%g3]
.L1:
  nop
  mov           $R, %o7
```

The problem with this snippet is that it needs to have many free registers available, besides those available from compiling without the application registers. Those can sometimes be found with the help of liveness analysis, but far too often, especially in the heavily optimized programs, it is

hard to always find free registers. This leads to the need to spill registers on to the stack, which in turn slows down the program considerably.

5.1.2 Floating-Point Load Snippets

Below the floating-point load snippet used by the proof-of-concept system can be seen . The purpose of this snippet is to handle floating-point loads in much the same way as integer loads are handled.

```
!ORIG_LOAD: ld [%o7], %f7

  mov          %o7, $R
  add          %o7, 892, %l0    ! spill %o7
  ld           [%o7], %f7       ! ORIG_LOAD
  fcmps        %fcc1, %f7, %f7
  fbe,pt       %fcc1, .L1
  nop
  srl          %l0, 29, %g3
  sub          %g3, 4, %g3
  brnz,pn      %g3, .L1
  call         DSZOOM_mem_load  ! %o7 changes
  mov           %l0, %g3        ! delay slot
  ld           [%l0], %f7
  sth          %g4, [%g3]
.L1
  mov          $R, %o7
```

The main problem with this snippet is that it uses the register %fcc1. It is not always sure that this register is free for us to use. And if not, you have to spill it to memory, getting the same disadvantages as for the integer load snippet.

5.1.3 Store Snippets

The third kind of snippet used is the store snippet. This snippet handles both integer and floating-point stores. How the original snippet looked is shown in Figure 9.

This snippet has, like the integer load snippet, to have at least two extra free registers available. As mentioned before this is seldom true and we have to spill to the stack.

5.2 New Optimizations

First of all we used the snippets that was used earlier to show that instrumenting with the new tool gave the same result as before. When this was done, we pretty quick realized that the snippets used and some of the aspects of the runtime-system was not as fine tuned as possible. The changes we have implemented are described in detail below.

```
!ORIG_STORE: st %o0, [%l3 + 340]

  mov          %o7, $R
  add          %l3, 340, %g3           ! g3 = eff addr
  srl          %g3, 29, %g4
  sub          %g4, 4, %g4
  brnz,pn      %g4, THIS_IS_LOCAL_ST
  srl          %g3, 6, %g4
  sethi        %hi(0x9b000000), %g3    ! g3 = %MTAG_ADDR_REG
  add          %g4, %g4, %g4
  add          %g3, %g4, %g3
  ldstub       [%g3], %g4
  brnz,pn %g4, 0x2a384
  nop
  ldub         [%g3 + 1], %g4
  sub          %g4, %g2, %g4
  brz,pn       %g4, I_AM_IN_MSTATE
  mov          %g3, %g4                ! g4 = %MTAG_ADDR_REG
  add          %l3, 340, %g3
  call                DSZOOM_mem_store ! %o7 changes
  nop
I_AM_IN_MSTATE:
  st           %o0, [%g3]              ! Orig ST
  sth          %g2, [%g4]              ! Unlock MTAG
  ba,a    END
THIS_IS_LOCAL_ST:
  st           %o0, [%g3]              ! Orig ST
END:
  mov          $R, %o7
```

Figure 9: The store snippet.

5.2.1 Reducing the Number of Instructions and the MTAG Size

We noticed that when running programs compiled on the highest optimization level, the liveness analysis seldom found enough free registers for the old snippets. Since EEL could not be used on optimized code this problem was not found earlier, since in un-optimized code free registers were easy to find. This lead to us having to spill the contents of registers to memory. The effect of this was often two extra memory accesses in every snippet, a high price to pay.

The registers we had available, without having to spill, was the registers reserved for us by the compiler when we used the no%appl flag and the registers found by the liveness analysis. Since the second set of registers often were none, we had to rely heavily on the reserved registers, that is registers %g2, %g3 and %g4. Those three registers proved to be sufficient, if we rewrote the snippet in a smarter way. No actual changes to the way the snippet worked was done during this, only in the way which registers was used and when.

Another thing we also realized was that having one byte for directory and one byte for MTAG lead to more cache pollution than was necessary. Cache pollution is when we store a lot of our runtime-system things in the cache, thereby kicking out things put in the cache by the actual program. By changing the runtime-system so that one byte could keep both the bits for the directory as well as the MTAG we not only lowered cache pollution, but also was able to remove one load instruction from each store snippet.

At this point, we still had not done any changes to the behavior of the DSZOOM system. All we had done was to duplicate the things the proof-of-concept system did, but in a slightly more efficient and robust way.

5.2.2 Straightening Out the Code

It is common that most of the code in the snippet will, in most cases, never be executed. It only takes up place in the instruction cache. Thus, the idea was to “take away” the part of the code that was not used for most of the time and in this manner produce a “straighter” execution path. By this we mean that it usually just continues and seldom has to branch to the code not so often used. This leads to less extra code having to be present in the instruction cache and therefore lowering cache pollution.

This was implemented so that we broke each snippet into two parts. For the integer load snippet it worked like this: The first of these parts just performed an access check. If it was correct it just continued. Since this in most of the cases is true we only have four extra instructions inserted for each load. But when this was not true the branch at the end of the snippet was taken to the other part of the snippet. In this part the range check and call to the coherence routine was executed. This other, not so often executed, part of the snippet was then placed at the end of the routine being instrumented. There all of the so called other parts of the snippets were placed together and only brought into the cache when needed.

The other snippets are divided into halves in the same way as the integer load snippet and only

the part that is most often executed is inserted next to the instrumented instruction. An example of how the integer load snippet looks is given in Figure 10.

```
IntLoad
#
!This is the fast path
!The code is placed next to the instrumented instruction
  *1 add $1,$2,%g3
   $I
   and $3,255,%g4
   sub %g4,255,%g4
   brz,pn, %g4,.LY$L
   *1 nop
   *2 add $1,$2,%g3
.LQ$L:
#

IntLoad2
#
!This is the slow path
!The code is placed at the end of the procedure
.LY$L:
   srl %g3,28,%g4
   sub %g4,8,%g4
   brnz,pt %g4,.LQ$L
   nop
   save %sp,-112,%sp
   mov %y, %l0
   mov %g1,%l1
   mov %ccr,%l2
   mov %fprs, %l3
   mov %g5,%l5
   call  DSZOOM_mem_load_real!
   mov %g3,%g6
   mov %l0,%y
   mov %l1,%g1
   mov %l2,%ccr
   mov %l3,%fprs
   mov %l5,%g5
   restore
   $D [%g6],$3
   stb %g4,[%g3]
   ba .LQ$L
   nop
#
```

Figure 10: Here the two parts of the snippet for integer loads are given.

The more detailed results achieved by this kind of “straighter” code is given in section 6.3, where it is shown that this actually leads to improved execution time of the instrumented programs and that it in average gives a speedup of 5%.

5.2.3 Removing Local Memory Accesses

It can be hard at instrumentation time to know what loads and stores are really global. If we do not know this there is a risk that we instrument even some local loads and stores and this

will lead to a bigger overhead than is necessary. The first and easiest thing that is done to avoid instrumenting some of those local loads and stores is to look at the arguments given to the instruction. The idea is to find loads and stores where there exists an argument that is a constant. Those can be found via the use of the instruction sethi. This instruction sets the highest 22 bits of a register to a constant. Then, using the register assigned 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: LD [%g1+lo(num_rows)], %g5. Therefore, choosing not to instrument loads or stores where the argument includes a register plus a lo construct, leads to avoiding some of the local memory accesses otherwise instrumented. This strategy has also been implemented in the instrumentation tool.

But this simple approach sometimes is not enough. When profiling all the programs used in the SPLASH-2 suite we found that for some of them we still instrumented a great number of local loads and stores. This obviously gives a larger overhead than necessary, since each instrumented load and store gives rise to extra instructions executed. For the programs that were worst at this there was a larger number of local loads/stores instrumented than global loads/stores.

Realizing this we now thought that we wanted to see how good our results would get if we could totally avoid instrumenting local memory accesses. (This could for example be done if you moved the instrumentation and did it in a compiler.) The way we could check this was to cheat some. First we ran the program with a special snippet that for each load or store called a subroutine in C that checked whether or not this was a global load/store by looking at the address given as argument. The routine then wrote both an unique ID given to this instruction as well as the information if it was global or not to an array specially allocated to store this. After the program was run, another routine wrote all the information stored in the array to a file called annotate_data. Once this file exists the program is instrumented again, but this time given the annotate_data file as an argument. The instrumentation tool then parsed this file into a HashTable. And now for every load and store we wanted to instrument we first looked in the HashTable to see if the instruction's ID was registered as a global or a local memory access. And the instrumentation was only done if it was only global. This way only instructions that were truly global were instrumented and no unnecessary overhead were present. The proof of the fact that none local instructions were instrumented was evident in the profiling output from a second run.

5.2.4 Optimization of the Store Operations

Using the additional, previously unused, free register available to us when compiling with the no%appl flag on SPARC V8plusa we have come up with a proposal to even further optimize some programs. The idea we had was that locking and unlocking the cache-line corresponding to each unique MTAG every time we wanted to do a store to it was sometimes an unnecessary, time-consuming task. If the program was doing two consecutive stores to the same cache-line it would be possible to just keep the lock some extra time, thereby avoiding one lock and one unlock. Of course we would have to insert some extra instructions to be able to do this, but

if there was even more consecutive stores to the same cache-line, there could be something to benefit from this. So we wrote a snippet that simply wrote the ID of the MTAG to a file and when analyzing those files for all the different programs we found that in some programs, most notably LU, there were up to 32 consecutive stores to the same cache-line. Obviously we could benefit from this spatial locality.

What we then did was to use the currently unused register, %g2, to hold a unique MTAG ID corresponding to each MTAG. Instead of releasing the lock for the cache-line at the end of the snippet, we kept the lock and saved the ID in the %g2 register. Then when finding a new instrumented store, the first thing we did in the snippet was to check if the MTAG ID of this new store was identical to the one saved in %g2. If it was we just carried on and executed the store like it was a local store, since we already had all the rights to it. If we on the other hand encountered a store with a different ID we instantly released the previous MTAG and then continued as usual with acquiring the lock for this new MTAG and so on. Also this time we skipped to release the lock at the end and again saved the MTAG ID. Since just locking like this and never releasing might easily lead to deadlocks we also changed a few things in the run-time system. At the end of each barrier and global lock in the program we inserted code to release the lock currently held by the MTAG indicated by the MTAG ID in register %g2. This was inserted into the run-time system, but we also had to do one minor change to the source-code of the instrumented program itself. This was that after all processes in the program had finished and reached the final stage, we also there inserted the same code to release the lock held by %g2. This lead to us being able to run and finish the programs. Figure 11 depicts this new store snippet used to implement this.

The effect of this optimization is that we delay the time before the new value written by the store is available globally. But by doing this we can effectively write as many times we want to the same cache-line without having to acquire and release the lock for it, as long as there are no writes to other cache-lines in the meanwhile. The result obtained by this method are of two kinds: For some programs (LU-cont, LU-non-cont and FFT) we cut the overhead from the instrumentation. For all of the other programs we found no or a very small cut in overhead. The last thing being true if there were no or very few local stores instrumented, i.e., we had removed most of the static stores. If not, there sometimes was an increased overhead by applying this method. The exact numbers we got are shown in section 6.

Some simulations were also done to see how much could be gained from this method in theory. What we did was to calculate the average number of consecutive times that a program wrote to the same cache-line. This was repeated for different cache-line sizes and also with different number of registers to check if a cache-line had been written to just previous. There we found that bigger cache lines gave a greater number of consecutive stores to the same cache-line in average. We also found that by using two registers to keep track of the cache-lines that we could write to without acquiring the MTAG again we got higher numbers of consecutive writes. To see the actual results for this, both with one and with two store-buffer registers, see Figure 12 and Figure 13.

```
Store
#
!Fast path, the code executed here is short
!Here is the code executed when the IDs are the same
   *1 add $2,$3,%g4
   *1 srl %g4,6,%g3
   *2 srl $2,6,%g3
   sub %g3,%g2,%g3
   brnz,pt %g3,.LY$L         !If same CacheID goto MSTATE
   nop
.LQ$LMSTATE:
   $I
#
Store2
#
!Slow path, more code, but seldom executed
!This code is only executed when the IDs are not the same
.LY$L:
   !First range check
   srl %g2,22,%g3
   sub %g3,8,%g3
   brnz %g3,.LQ$LFALSE_MTAG
   nop
   !Release prev MTAG code, %g2 == old CacheID
   sethi %hi(0x8dc00000),%g4
   add %g4,%g2,%g4            !%g4 == PREV MTAG
   stb %g0,[%g4]              !Release and update old MTAG
.LQ$LFALSE_MTAG:
   add $2,$3,%g4              !Range check
   srl %g4,28,%g3
   sub %g3,8,%g3
   brnz,pn %g3,.LQ$LMSTATE
   srl %g4,6,%g2              !%g2 == current CacheID
   sethi %hi(0x8dc00000),%g4
   add %g4,%g2,%g4            !%g4 == NEW MTAG
.LQ$LPREV:
   ldstub [%g4],%g2
   sub %g2,255,%g3
   brz,pn %g3,.LQ$LPREV
   nop
   add $2,$3,%g3
   mov %g2, %g4
   brz,pn %g2,.LQ$LMSTATE     ! Check if in MSTATE
   srl %g3,6,%g2              !%g2 == current CacheID
   !Here the call to the coherence routine is done
   ba .LQ$LMSTATE
   nop
#
```

Figure 11: Here the two snippets for the optimization of delayed stores are shown.

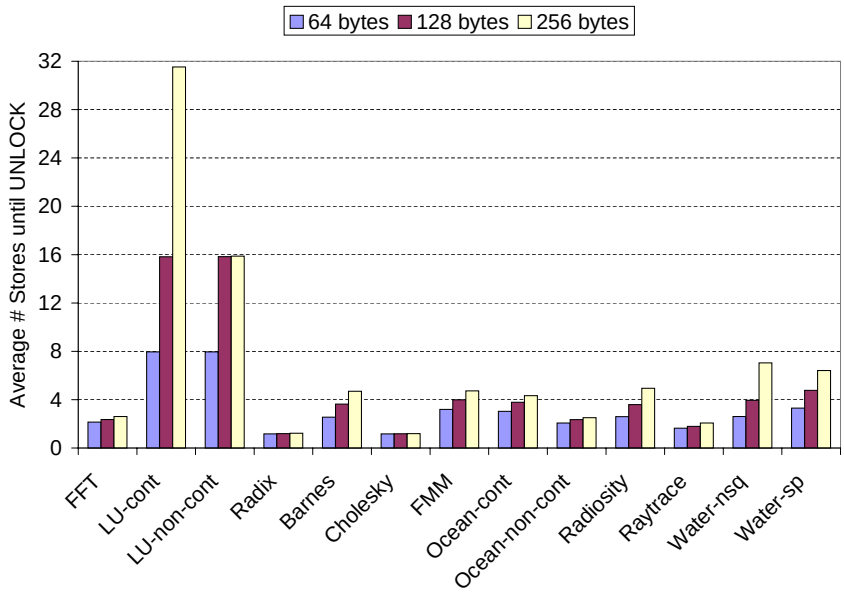


Figure 12: The average number of consecutive stores to the same cache-line for different cache-line sizes, using one store-buffer register.

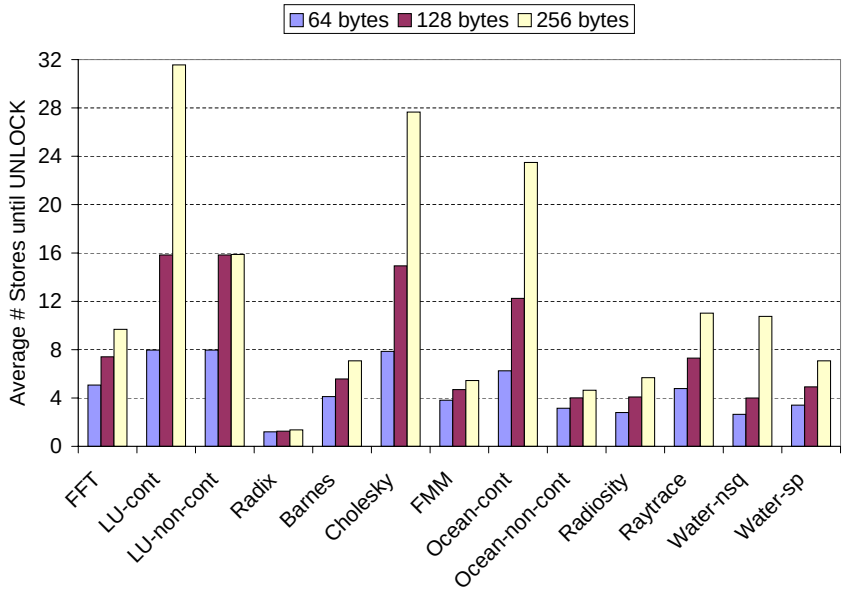


Figure 13: The average number of consecutive stores to the same cache-line for different cache-line sizes, using two store-buffer registers.

6 Performance Study

This section describes experimental setup, applications used in this study, and finally, we present a DSZOOM performance overview for this new instrumentation tool.

6.1 Experimental Setup

All the experiments done here are performed on the same system used to test the original implementation of DSZOOM (using EEL). As compiler we have used Sun's CC Forte version 6.2 instead of the GNU gcc compiler used before.

The hardware used is a Sun Enterprise E6000 SMP. The server has 16 UltraSPARC II (250 MHz) processors and 4 Gbyte uniformly shared memory with an access time of 330 ns (lm-bench latency) and a total bandwidth of 2.7 Gbyte/s. Each processor has a 16 kbyte on-chip instruction cache, a 16 kbyte on-chip data cache, and a 4 Mbyte second-level off-chip data cache.

The hardware DSM numbers have been measured on a 2-node Sun-WildFire built from two E6000 nodes connected through a hardware-coherent interface with a raw bandwidth of 800 Mbyte/s in each direction [HK99]. The Sun-WildFire has been configured as a traditional non-uniform memory architecture (NUMA) with its data migration capability activated while its coherent memory replication (CMR) has been kept inactive. The Sun-WildFire access time to local memory is the same as above, 330 ns, while accessing data located in the other E6000 node takes 1700 ns (lm-bench latency). The E6000 and the Sun-WildFire are both running a slightly modified version 2.6 of the Solaris operating system.

DSZOOM-WF system runs in user space on the Sun-WildFire with its data migration and the CMR data replication kept inactive.

6.2 Applications

To test the performance of this new DSZOOM we have used the well-known scientific workloads from the SPLASH-2 benchmark suite [WOT+95].

The sizes of the data-set used and the uninstrumented uniprocessor-execution times are presented in Table 1. The reason why we cannot run Volrend is because of the global variables used as shared. It should be possible to manually modify this application to get rid of this problem. We began all measurements at the start of the parallel phase to avoid DSZOOM-WF's run-time system initialization.

6.3 Performance Overview

As an explanation to what can be gained by running optimized code compared to unoptimized code, Table 1 shows the sequential execution times for all the programs in the SPLASH-2 bench-

mark, compiled both without optimization, using the -xO0 flag, and with the highest optimization, with the -fast flag. There it is seen that the speedup is over 4 times in average for those programs. Therefore much is to be gained by being able to instrument this kind of programs, even if this leads to larger instrumenting overhead (which will be seen later).

Program	Problem Size	Seq. Time -xO0 [s]	Seq. Time -fast [s]
FFT	1,048,576 points (48.1 MB)	14.29	3.18
LU-Cont	1024×1024, block 16 (8.0 MB)	66.61	13.56
LU-Non-Cont	1024×1024, block 16 (8.0 MB)	80.30	30.56
Radix	4,194,304 items (36.5 MB)	30.95	6.67
Barnes-Hut	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-Cont	514×514 (57.5 MB)	46.76	14.61
Ocean-Non-Cont	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 both the -xO0 and the -fast flag.

There are some initial limitations that we impose on our DSZOOM system. By compiling with the -xregs=no%appl flag we take away three registers that the compiler otherwise could have used to try and optimize the code even further. And also when our instrumentation tool lifts out loads and stores from the delay slot we work against optimizations that the compiler already has done. In Table 2 we show the results of those two initial slow-downs. There it is shown that exclusion of the application registers does not really effect the execution time, but emptying the delay slots of loads and stores slows the programs down with in average 3%.

Now we are ready to see how the instrumented programs perform on a single processor. Table 3 shows the performance for the instrumented programs for two different kind of snippets. The first snippet is just the normal, somewhat improved snippet that is the same as in the old DSZOOM system (described in section 5.2.1). The other 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 5.2.2). For both snippets execution times and overheads are given. We can see that straightening out the execution path actually is an optimization and the relative speedup between the two techniques are given in the last column. The percentage of statically replaced loads and stores are also shown in this table. Those are quite high and one thing to note here is that this leads to a rather large number of local loads and stores being instrumented for some programs. This leads to extra overhead.

Program	With Appl. Regs	Without Appl. Regs	Overhead	With Empty Delay Slots	Overhead
FFT	3.36	3.37	1.00	3.18	0.95
LU-Cont	13.15	13.27	1.01	13.62	1.04
LU-Non-Cont	29.71	30.16	1.02	30.61	1.03
Radix	6.96	6.96	1.00	6.66	0.96
Barnes-Hut	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-Cont	15.19	15.17	1.00	14.66	0.97
Ocean-Non-Cont	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 next thing we will see is how this new DSZOOM system perform when run in parallel. Table 14 shows the execution times for several different configurations using 8 CPUs. There are four different configurations that we have tested, two using DSZOOM and two not using DSZOOM. The difference between the two DSZOOM system is the following:

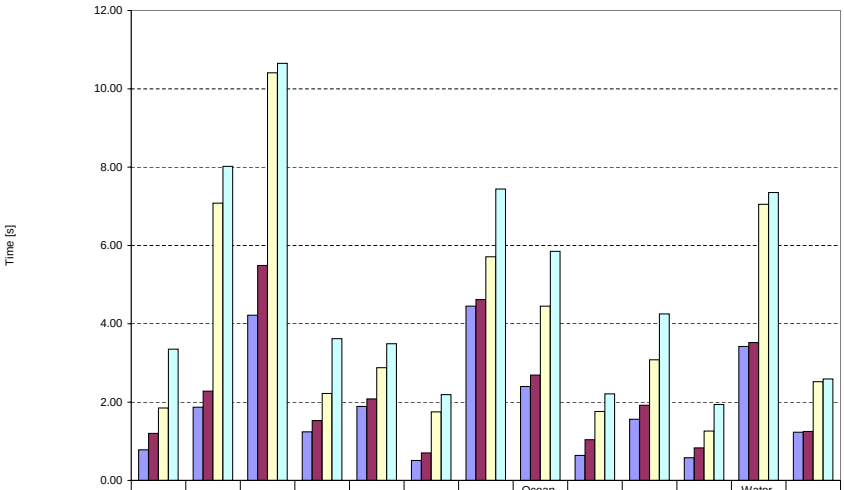
Single-node DSZOOM-WF. This is a system without any inter-node communication. All the processes are executed in the same node.

2-node DSZOOM-WF. This configuration is a “real” DSZOOM-WF implementation. Here both memory and processes are physically distributed across both nodes.

When it comes to the two non-DSZOOM configurations the difference between them are that one is just run on one Sun Enterprise E6000 and the other one is run on two Sun Enterprise E6000 as a 2-node CC-NUMA. The same things also apply to Table 15, but here instead 16 CPUs are used.

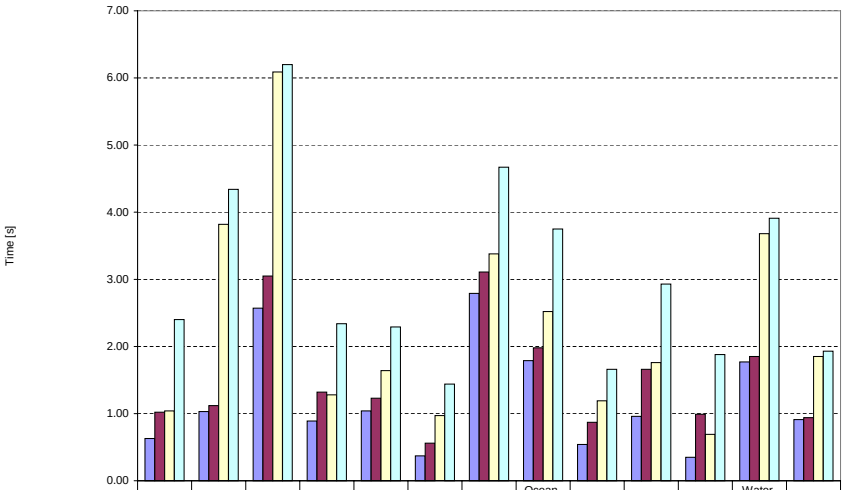
Program	Uniproc Time [s]	% Loads Replaced	% Stores Replaced	Ordinary Snippet	Straight Code (Section 5.2.2)	Overhead Straight Code	Relative Speed-up
FFT	3.18	50.2	45.1	10.19	9.93	3.12	0.97
LU-Cont	13.56	40.6	33.2	51.74	51.19	3.78	0.99
LU-Non-Cont	30.56	43.0	36.1	68.18	66.80	2.19	0.98
Radix	6.67	32.3	29.1	10.95	10.64	1.60	0.97
Barnes-Hut	13.28	59.1	64.4	21.14	20.13	1.52	0.95
Cholesky	3.45	60.5	43.8	12.33	11.83	3.43	0.96
FMM	25.03	64.8	51.7	42.17	37.45	1.50	0.89
Ocean-Cont	14.61	63.5	69.9	30.91	N/A	N/A	N/A
Ocean-Non-Cont	3.72	41.7	65.6	8.03	N/A	N/A	N/A
Radiosity	11.10	69.8	64.0	23.32	21.99	1.98	0.94
Raytrace	3.89	63.5	55.6	10.60	9.01	2.32	0.85
Water-nsq	26.07	42.6	32.6	51.14	52.02	2.00	1.02
Water-sp	7.76	37.2	27.5	16.09	14.96	1.93	0.93
Average		**51.4**	**47.6**			**2.28**	**0.95**

Table 3: Here the effect of the optimization described in section 5.2.2 is shown. First the execution times for the uninstrumented programs are shown. Then the number of replaced loads and stores when instrumenting are given. Then the execution times for programs instrumented with the ordinary snippet and the optimized snippet are given. Finally the relative speedup for the optimized snippet compared to the ordinary snippet is shown.



	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

Figure 14: Parallel performance for 8 CPUs. Execution times in seconds for Sun Enterprise E6000, 2-node Sun-WildFire, single-node DSZOOM-WF and 2-node DSZOOM-WF.



	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 15: Parallel performance for 16 CPUs. Execution times in seconds for Sun Enterprise E6000, 2-node Sun-WildFire, single-node DSZOOM-WF and 2-node DSZOOM-WF.

6.4 Performance When Removing Local Memory Accesses

Since there are many loads and stores that are instrumented that actually are local, we have in section 5.2.3 presented a way to reduce or totally avoid instrumenting those local loads and stores. The results obtained when using this technique is given in this section. As a first result we can see in the table below that the number of statically replaced loads and stores are drastically reduced.

Program	% Loads Replaced	% Stores Replaced
FFT	15.1%	17.8%
LU-Cont	13.5%	16.0%
LU-Non-Cont	13.1%	15.1%
Radix	8.8%	16.6%
Barnes-Hut	19.8%	21.4%
Cholesky	16.8%	13.7%
FMM	N/A	N/A
Ocean-Cont	39.7%	52.5%
Ocean-Non-Cont	31.9%	50.7%
Radiosity	N/A	N/A
Raytrace	20.8%	14.8%
Water-nsq	17.1%	15.4%
Water-sp	19.0%	12.0%
Average	**19.6%**	**22.4%**

With this technique the performance is improved and the instrumentation overhead is cut. When this is combined with the technique described in section 5.2.4the performance is even further trimmed. Still the overhead on optimized code is large compared to the overhead that the earlier DSZOOM system got on unoptimized code. But on unoptimized code, using the new techniques the overheads are comparable to those obtained before, and sometimes even better. The execution times and overhead for the optimizations techniques described in sections 5.2.3 and 5.2.4 are given in Table 4, with maximum optimization, and in Table 5, without any optimization.

The same hardware configurations have been used to get the parallel results in this case too, with the exception that we have only used the 2-node configurations, i.e., the 2-node CC-NUMA and the DSZOOM-WF on two nodes. But this time we have tried the DSZOOM system with two different kinds of optimizations, first only removing local memory accesses and then with both that and the optimization for stores described in section 5.2.4. The results from those measurings are given in Table 16 and Table 17.

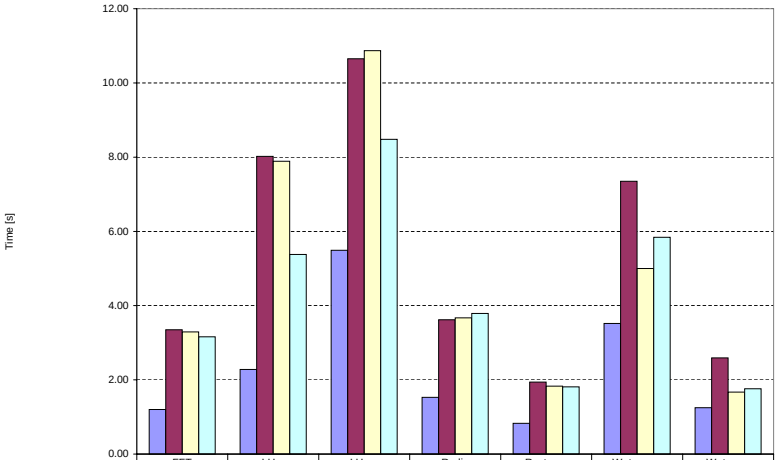
Finally, we give a comparison between the performance of a DSZOOM system that uses the new tool and the new optimizations available, and a earlier DSZOOM system that uses EEL. There we can see that running optimized code, even if it gives larger instrumentation overhead, still runs faster than unoptimized code. The results are shown in Figure 18.

ALL -fast Program	Uniproc. Time [s]	Unsliced Instr Uniproc. Time	Unsliced Overhead	Instr. Uniproc. [s] (Section 5.2.3)	Overhead (Section 5.2.3)	Instr. Uniproc. [s] (Section 5.2.4)	Overhead (Section 5.2.4)
FFT	3.2	10.0	212.9%	9.7	206.3%	8.4	164.2%
LU-cont	13.6	51.1	276.8%	51.4	279.3%	33.5	147.1%
LU-non-cont	30.6	67.2	119.8%	67.9	122.3%	50.5	65.2%
Radix	6.7	10.6	59.5%	10.6	58.9%	11.7	75.4%
Barnes	13.3	20.0	50.8%	15.3	15.1%	15.8	19.0%
Cholesky	3.5	11.8	242.6%	11.8	243.2%	15.36	345.2%
FMM	25.0	38.4	53.3%	N/A	N/A	N/A	N/A
Ocean-cont	14.6	31.4	115.1%	31.6	116.1%	27.9	90.9%
Ocean-non-cont	3.7	8.1	117.5%	8.0	114.5%	7.7	106.2%
Radiosity	11.1	22.1	98.8%	12.4	11.7%	12.8	15.7%
Raytrace	3.9	9.0	131.9%	6.6	68.9%	6.5	67.1%
Water-nsquared	26.1	52.4	100.8%	33.3	27.7%	31.7	21.6%
Water-spatial	7.8	14.8	90.9%	9.4	20.7%	9.2	18.6%
Average			**128.5%**		**107.1%**		**94.7%**

Table 4: The performance for the optimizations described in Sections 5.2.3 and 5.2.4. First with only the optimization in Section 5.2.3 and then with both optimizations. Compiled with the -fast flag.

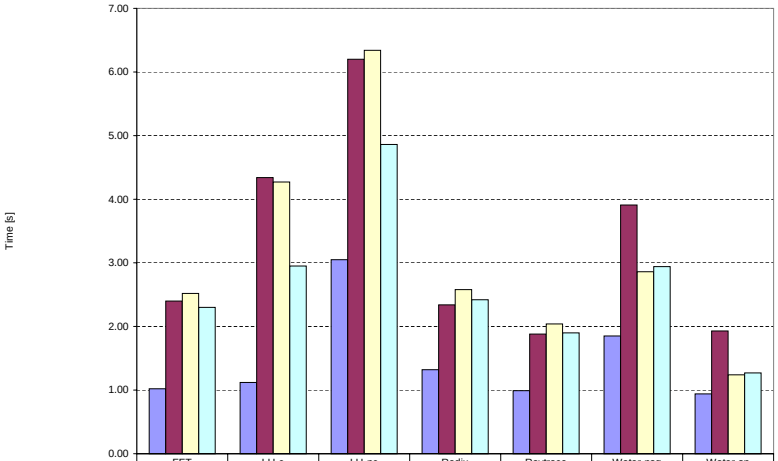
ALL -xO0 Program	Uniproc. Time [s]	Unsliced Instr Uniproc. Time	Unsliced Overhead	Instr. Uniproc. [s] (Section 5.2.3)	Overhead (Section 5.2.3)	Instr. Uniproc. [s] (Section 5.2.4)	Overhead (Section 5.2.4)
FFT	14.3	26.8	87.8%	27.2	90.1%	20.5	43.3%
LU-cont	66.6	149.2	123.9%	148.2	122.5%	94.0	41.1%
LU-non-cont	80.3	163.6	103.8%	163.1	103.1%	109.2	36.0%
Radix	30.9	36.5	17.8%	34.8	12.5%	37.0	19.6%
Barnes	57.0	73.1	28.2%	57.8	1.3%	57.4	0.7%
Cholesky	20.2	34.7	72.2%	36.6	81.2%	33.4	65.7%
FMM	117.6	144.2	22.6%	128.5	9.3%	127.6	8.5%
Ocean-cont	46.8	80.6	72.3%	77.9	66.7%	63.6	36.0%
Ocean-non-cont	18.3	27.7	51.3%	26.1	42.6%	23.3	27.1%
Radiosity	29.0	40.6	40.1%	33.8	16.5%	33.7	16.2%
Raytrace	11.3	26.0	130.7%	11.9	5.3%	11.9	5.7%
Water-nsquared	134.5	284.5	111.6%	187.3	39.3%	187.8	39.7%
Water-spatial	34.3	74.2	116.1%	48.7	41.9%	47.9	39.6%
Average			**75.3%**		**48.6%**		**29.2%**

Table 5: The performance for the optimizations described in Sections 5.2.3 and 5.2.4. First with only the optimizations described in Section 5.2.3 and then with both optimizations. Compiled with the -xO0 flag.



	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

Figure 16: Parallel performance for 8 CPUs with the optimization techniques described in section 5.2.3 and 5.2.4. Execution times in seconds for 2-node Sun-WildFire, 2-node DSZOOM-WF (a), 2-node DSZOOM-WF with the optimization from sections 5.2.3 (b), and 2-node DSZOOM-WF with the optimizations from both sections 5.2.3 and 5.2.4 (c).



	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 17: Parallel performance for 16 CPUs with the optimization techniques described in sections 5.2.3 and 5.2.4. Execution times in seconds for 2-node Sun-WildFire, 2-node DSZOOM-WF (a), 2-node DSZOOM-WF with the optimization from section 5.2.3 (b), and 2-node DSZOOM-WF with the optimizations from both sections 5.2.3 and 5.2.4 (c).

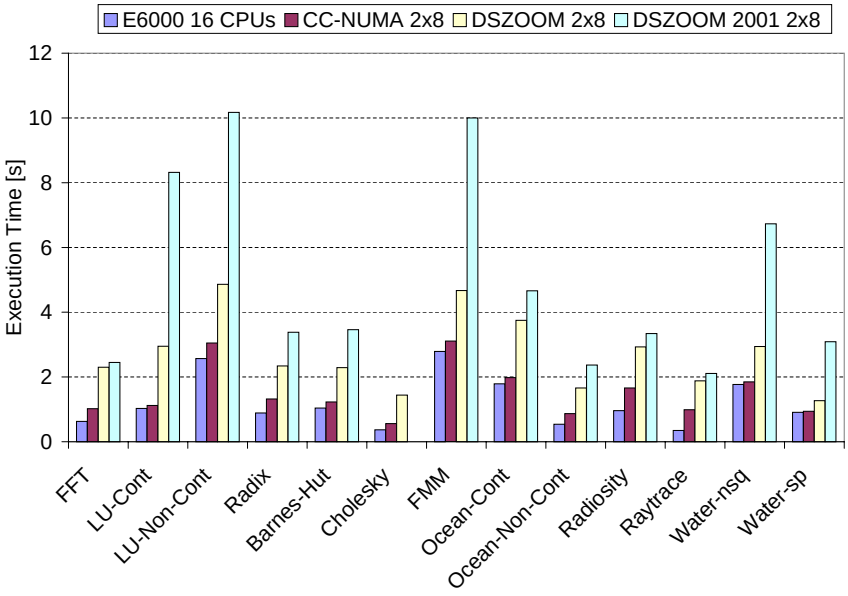


Figure 18: A comparison between a Sun Enterprise E6000, a 2-node Sun-WildFire, a 2-node DSZOOM-WF compiled with -fast, instrumented with the new tool and using the new optimizations, and a 2-node DSZOOM-WF (2001) compiled with -xO0 and instrumented with EEL.

7 Conclusions

As seen in the previous sections the overhead we get when we instrument programs compiled with the highest optimization is greater than the overhead gotten on programs compiled without optimization. The overhead goes from around 30% for the un-optimized programs to around 100% for programs with maximum optimization. Still the actual execution time is lower for the optimized programs than the un-optimized ones. So even if we do get a greater overhead when instrumenting optimized binaries compared to un-optimized binaries, in actual execution time we beat the old times for all programs tested. We get that the performance improvements range from 1.07 to 2.82 times (average 1.73).

It seems that by applying the store-buffer register optimization described in section 5.2.4 much can be gained in terms of performance for a certain type of applications.

Also, for different programs in the SPLASH-2 benchmark sometimes different snippets need to be used. Using this new instrumentation tool writing and changing snippets is an easy task.

Acknowledgments

Many thanks go to the people in the UART-group (UPPSALA ARCHITECTURE RESEARCH TEAM) at Uppsala University that helped me with this thesis. Especially my supervisors Zoran Radovic, who have given me so much of his time, and Erik Hagersten. I would also like to thank Marika Nestor for support during this time.

This thesis is supported in part by Sun Microsystems Inc. and the Parallel and Scientific Computing Institute (PSCI), Sweden.

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. The Press Syndicate of the University of Cambridge, 1998.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP'91)*, pages 152–164, October 1991.

[CERL01] C. Cifuentes, M. Van Emerik, N. Ramsey, and B. Lewis. *The University of Queensland Binary Translator (UQBT) framework*, December 2001. Documentation from the UQBT open source distribution, availible from http://www.itee.uq.edu.au/cms/uqbt.html.

[HK99] 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.

[KCDZ94] 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.

[Kel95] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Computer Science, Rice University, January 1995.

[Li88] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing (ICPP'88)*, volume II, pages 94–101, August 1988.

[LS95] 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.

[RH01a] Z. Radović and E. Hagersten. DSZOOM – Low Latency Software-Based Shared Memory. Technical Report 2001:03, Parallel and Scientific Computing Institute (PSCI), Sweden, April 2001.

[RH01b] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of Supercomputing 2001*, Denver, Colorado, USA, November 2001.

[SFH+96] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lucas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[SFH+98] 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.

[SFL+94] 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.

[SG97a] D. J. Scales and K. Gharachorloo. Design and Performance of the Shasta Distributed Shared Memory Protocol. In *Proceedings of the 11th ACM International Conference on Supercomputing*, July 1997. Extended version available as Technical Report 97/2, Western Research Laboratory, Digital Equipment Corporation, February 1997.

[SG97b] 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.

[SGT96] 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.

[Sun01] Sun Microsystems. *C User's Guide Forte Developer 6 update 2*, 2001.

[Sun02] Sun Microsystems. *ABI Compliance and Global Register Usage in SPARC V8 and V9 Architecture*, 2002. http://soldc.sun.com.

[Web02] WebGain. *Java Compiler Compiler Grammar File Documentation*, June 2002. Documentation for the Java Compiler Compiler Grammar File, availible from http://www.webgain.com.

[WOT+95] 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.