Using Explicit Spill and Fill Instructions to Increase ILP and Improve Memory Bandwidth

Mattan Erez
Brian Towles
David Lie
Dean Liu
Shuaib Arshad

May 24, 2000

1.0 Introduction

The majority of modern superscalar processors use *Reduced Instruction Set Computer* (RISC) type instruction sets. However, their hardware design is no longer aligned with the RISC philosophy and is getting more and more complex. The ISA originally designed for simplicity now causes unnecessary complication for the compiler and the architecture: hardware ignores many of the optimizations performed by the compiler and the compiler "tricks" the hardware to achieve desired results. For example, a major step of compilation is register allocation and spill-code generation. Upon receiving the compiler's carefully constructed register assignment, the hardware promptly renames the registers into unique hardware identifiers, throwing away much of the compiler effort. Ironically, this renaming operation proves to be a bottleneck of modern processors since it requires serial treatment of all instructions. This example and similar problems illustrate an *ISA barrier* to efficient superscaler execution.

A key element in overcoming the ISA barrier is to allow the compiler to effectively communicate its plan of execution to the hardware. Returning to the spill-code generation example: when the compiler runs out of logical registers, it spills the contents of a logical register to memory using a memory store. If there are more physical registers than the software addressable architectural registers, then the process may rename some of the registers during run time. As a result, this memory reference issued by the compiler may be completely unnecessary because the processor has already renamed the logical register to some physical register and there are other physical register that the processor can rename to. For processor with a large number of physical registers not available to the compiler, it may be more efficient for the compiler to state its intent of spilling, then to issue an unnecessary memory reference. Table 1 lists the number of registers in some modern processors.

Table 1: Logical Register vs. Physical Register

Processor	ISA Register (Int / FP)	Physical Register (Int / FP)
MIPS R10000	32 / 32	64 / 64
Alpha 20264	32 / 32	80 / 72
Pentium Pro	4 / 8	40

The large gap between the number of logical registers and physical registers exacerbates the miscommunication between the compiler and the processor. In an attempt to clarify the compiler's intent and to take advantage of the growing number of physical registers, we propose a pair of instructions to use in the spill-code generation. This addition to the ISA is not tailored to any specific hardware. Instead, these instructions are general enough such they allow more efficient operations of software on any superscaler processor. The mnemonic and format of these instructions are given below:

```
spill Rx, spill_ptr        (substitution of store)
fill  Rx, spill_ptr        (substitution of load)
```

Ideally, there is no address and the HW has enough resources to handle all spills. But if the hardware actually runs out of resources it will need a complex negotiation with the OS (or runtime system) to get memory, and also very complex mechanisms and algorithms to handle it. To simplify the hardware, we introduce a spill pointer to keep track of a stack of relative addresses. The most obvious reason for using a stack is when there is a recursive call.

It is possible for a spill to have multiple fills if the data is not modified in between the fills. Therefore it is important to track these fills so the hardware will know when it can reclaim the register. There are two ways to track these fills: a. issue spill and fill in pairs or b. have an explicit fill-kill instruction to indicate the last fill. If the compiler issues the spill and fill in pairs, then each fill implicitly implies that there will be no other fills for this spill. On the other hand, if a fill-kill instruction is used, then the hardware must realize that the spilled register may have multiple fills until the last fill is encountered as a fill-kill.

Adding two explicit instructions to spill-code generation to take advantage of the physical registers is a more elegant solution than to increase the addressable space of the software. First of all, instruction word format dictates the number of logical registers, so the number of software addressability cannot be changed for existing ISA. Secondly, the spill and fill instructions allow a very flexible hardware implementation. The designer can simply choose to execute them as store and load instructions, or add controls to handle the spill code efficiently. This range of implementations provides more freedom to the hardware designer. A more detailed look at some of these options is presented in Section 4.

Figure 1 shows a pseudo code to help illustrate the use of the explicit spill and fill. Assume there are two logical registers, R_1 and R_2, and four physical register (P_1, P_2, P_3, and P_4). The first column in the figure is the line number, the second column is the pseudo instruction, the third column is the logical register that's being referenced, the fourth column shows the physical register that the logical register is mapped to or the action that would be taken if there is no spill/fill instructions and the last column shows what is actually done when spill/fill are supported.

1	use	R_1	P_1	
2	use	R_2	P_2	
3	*spill*	R_1	P_1 -> Mem	Tag P_1
4	ld	R_1	Mem -> P_3	
5	use	R_1	P_4	
6	*fill*	R_1	Mem -> P_1	Rename R_1 to P_1 Untag P_1

Figure 1. Example Code

Line 3 shows that the content of logical register, R_1 needs to be spilled. At this time, R_1 is being mapped to P_1. If the spill instruction is not supported, this instruction will turn into a memory reference and store the content of R_1 to the memory. However, if the processor supports the spill instruction, then it simply tags the physical register that is associated with R_1. In other words,

using spill instead of store delays the memory reference. If there are enough physical registers such that the front end can always rename, then spill delays the memory reference indefinitely. However, if there is not enough physical registers, then the spill is executed only when it is absolutely necessary to reclaim one physical register.

Similarly on a processor that supports fill, a fill instruction results in the untag of a previously tagged physical register, as seen in line 6. As in the case with spill, if there are enough physical registers, the reference to load P_1 is handled in the front end if it does not need to go to the memory.

1.1 Hypothesis

We hypothesize that using explicit spill and fill instructions increase ILP by reducing the produce-store-load-use false dependences and improve memory bandwidth by reducing the superfluous stores and loads.

1.2 Organization

The rest of the report is organized as follows. Section 2 will detail our simulation method by first describing the base model, then detailing our modifications and the benchmarks. The results of the simulations are reported in section 3. In section 4 we suggest some future work which is followed by our conclusions in section 5.

2.0 Simulation Method

In this section we discuss our simulation method. We start by describing our simulator, follow by a description of the two experiments we run to prove or disprove our hypothesis.

2.1 Simulator Model

In order to verify our hypothesis, we enhance the gcc compiler to annotate the spill and fill instructions, and modify the SimpleScaler simulator to recognize these annotations [1][2]. To add the spill and fill instructions, we annotate memory operations caused by register allocation failures and caller/callee register saving. These annotations are passed to the assembler and stored as a bit field in the instruction word. This differentiates the spill code from other store and load instructions. The SimpleScaler simulator needs to not only recognize these new instructions, but also perform the necessary accounting to gather the performance data.

The simulator ran the SPECint95 benchmarks to show the improvements [3]. For all the benchmarks, we skip the first 500M instruction and collect data on the next 100M instructions completed.

2.2 Experiments

2.2.1 ILP

To show the effect of adding the explicit spill and fill instruction on ILP, we first find the maximum ILP given a fixed window size without the explicit spill/fill instructions. Then we run the same set of benchmarks with the explicit instructions. To find the maximum ILP, we make a few assumption about the hardware. First of all, we eliminate the control dependencies by assuming perfect branch predictions. Then we remove the name dependencies by providing an infinite number of physical registers and renaming. All data dependencies are honored. The fetch unit has infinite bandwidth and perfect branch and target prediction while the execute unit has infinite hardware to resolve any resource conflicts. All computations have a 1-cycle latency except for loads which take 2 cycles. As a results of the infinite number of physical registers, the spill and fill never need to go to the memory and thus have zero latency.

We simulated three memory name dependency models:

a. No disambiguation (in order stores, loads wait until there is no prior store)
b. Load disambiguation (in order stores, loads wait until there is no store to the same address)
c. Perfect store and load disambiguation

2.2.2 Memory Reference

To measure the impact in memory references, we gather three sets of data:

a. number of dynamic spills and fills
b. dynamic instruction distance between a spill and its corresponding fill
c. maximum number of outstanding spills (spills that have not been filled between a spill-fill pair)

The first number gives us the percentage of dynamic spills and fills to stores and loads. This is a direct measurement of how many memory references can be saved. The second number indicates whether the references are likely to be found in the registers. And the third number shows how much extra register space is needed to hold spilled data.

3.0 Results

In this section we analyze the data collected from the experiments to prove or disprove our hypothesis. From the ILP experiment, we measure the total ILP and the average ILP across all benchmark to see how much, using explicit spill and fill instructions, affect the ILP. The total and average ILP are calculated as follow:

$$ILP_{total} = \Sigma(\text{Inst. Exec.})_{benchmark} / \Sigma(\text{Cycle Exec.})_{benchmark} \quad (1)$$

$$ILP_{average} = \Sigma(ILP)_{benchmark} / \#.\text{ of benchmark} \quad (2)$$

Although these two ILPs look similar, they actually measure different things. ILP_{total} shows the ILP if the set of benchmark is representative of all programs executing on the machine, whereas the $ILP_{average}$ shows an unweighted average improvement. In other words, ILP_{total} shows the

improvement if everyone runs all of the programs, and $ILP_{average}$ shows the improvement if each person runs one of the program. Figure 3 shows the ILP_{total} and $ILP_{average}$ under various instruction window sizes (32, 64, 128, and 256) with and without spill and fill instructions.

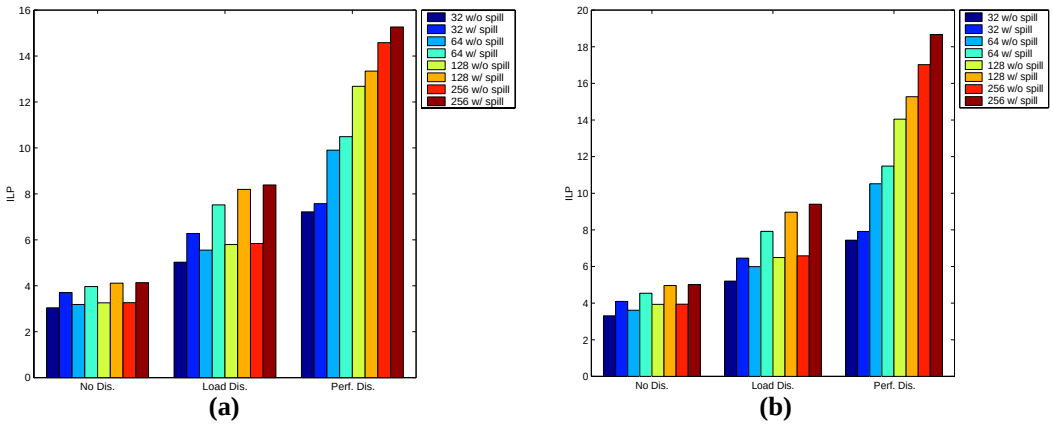

Figure 3. Total (a) and Average (b) ILP with 2-Cycle Load Latency and Different Disambiguation Policies

From Figure 3, we see that the total and average ILPs look very similar. This indicates that the benchmarks used in our simulation give the correct mix of applications and we can expect about 22% increase in ILP when explicit spill and fill instructions are used.

In addition to showing an improvement in ILP when spill and fill instructions are used, Figure 3 also illustrates a dramatic increase in ILP with the different disambiguation policies. This is because with no disambiguation, there are many false dependencies on waiting stores. While with load disambiguation only, the scheduling window tends to fill up with stores which limits the parallelism that can be utilized. Finally, the full disambiguation policy gives the full benefit of have larger windows.

We now explore how varying the load latency affects ILP in our simulation. Figure 4 shows that total ILP with only load disambiguation has the same trend across different load latencies. This means that the maximum ILP is not very sensitive to small variations in load latencies.

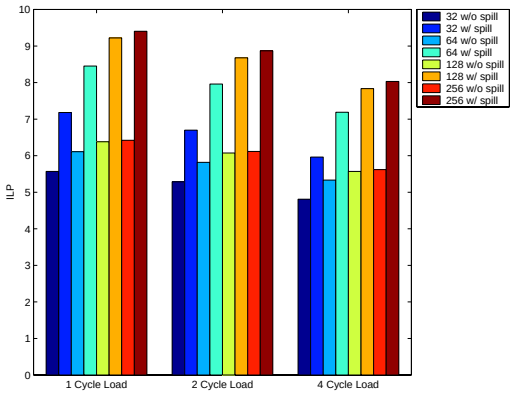

Figure 4. Total ILP with Load Disambiguation and Different Load Latencies

Finally, for our ILP simulation, we find that gcc is representative of the benchmarks that worked

well with the explicit spill and fill instructions (i.e., gcc, li, perl, and vortex), and ijpeg of the others (i.e., go, ijpeg, and m88ksim). Compress is unique in that it has no spills. Figure 5 shows the ILP of gcc and ijpeg with and without spill instructions.

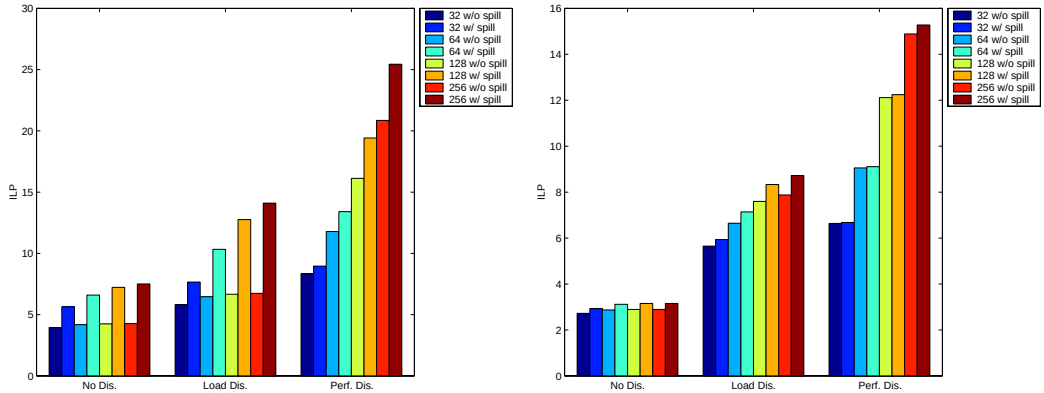

Figure 5. ILP for (a) gcc and (b) ijpeg with 2-Cycle Load Latency and Different Load Disambiguation Policies

Comparing the ILP with and without explicit spill and fill instructions shows that our proposed technique improves ILP by 8-100% in the case of gcc and 5-11% in ijpeg. Overall, using these explicit instructions increase the ILP by 22%.

We now look at the impact of using spill and fill on memory references by examining the data collected from the Memory Reference experiment. First, we measure the number of spills between a spill and fill pair. This gives an indication of how many available physical storage is needed if we have to service the fills. Figure 6a plots the percent of all fills versus the number of spills between a spill/fill pair. The figure shows that having a 64-entry register space for spilling only can capture about 92-100% of all the outstanding spills. This means that we can potentially eliminate all of the memory references used for spill code.

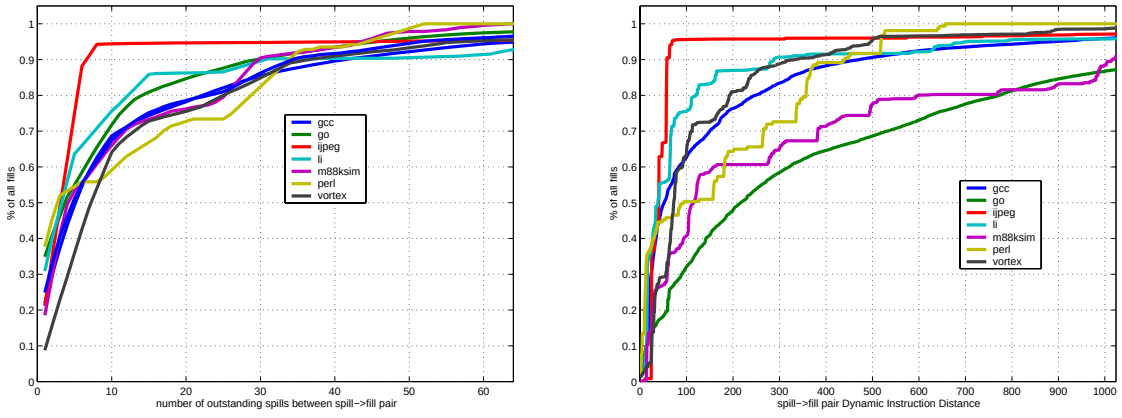

Figure 6. (a) Max. Number of Outstanding Spills between a Spill / Fill Pair
(b)Number of Dynamic Instructions between a Spill / Fill Pair

Figure 6b plots the number of dynamic instructions between a spill/fill pair. From this figure we

see that more than 85% of the spill will be filled within the next 1000 dynamic instructions. This means that the spilled data is needed fairly soon, so the hardware should refrain from moving the data to memory.

Having established that using the spill and fill instructions can potentially reduce the memory references, we now examine how much impact these explicit instructions actually have. Figure 7a plots the percentage of dynamic stores and loads that are converted to spills and fills. The percentage of spill ranges from 15% to 62%, and the percentage of fill ranges from 9% to 31%, with an average of 35% and 19% for spill and fill, respectively. This measurement indicates that a significant portion of the memory references are spill-code and can be eliminated. Originally, we thought that all of the stack operations can be converted to spills and fills. However, Figure 7a clearly indicates otherwise.

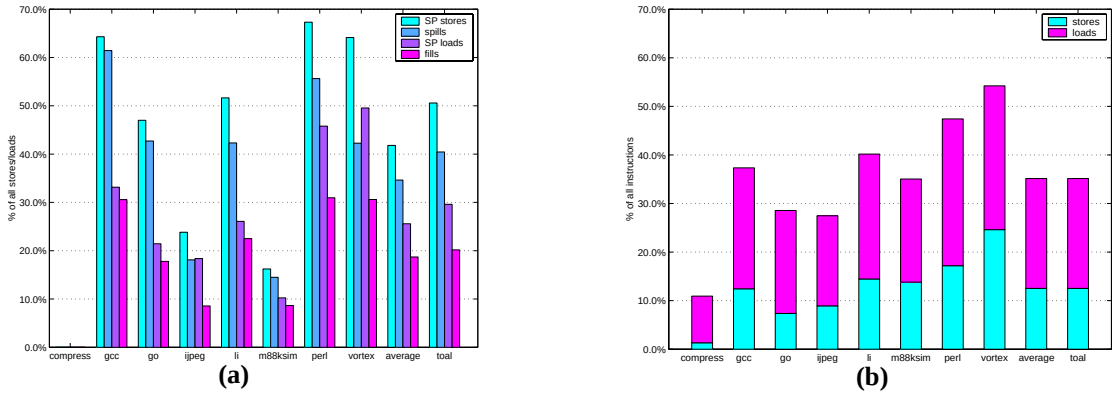

Figure 7. (a) Spill and Fill Statistics (b) Load and Store Statistics

To put the results from Figure 7a into perspective, the percentage of store and load in all of the dynamic instructions is plotted in Figure 7b. The data shows an average of 12% stores and 35% loads. These numbers indicates the memory reference reduction achieved by using spill and fill instructions is significant.

4.0 Future Work

In this section, we will describe several hardware options for exploiting the new information provided by the explicit spill/fill instructions. First let us detail the new instructions and elaborate on their properties:

spill *Rx, spill_ptr* - spill a register to the memory location spill_ptr
fill *Rx, spill_ptr* - fill a register from the location spill_ptr
setsip *immediate* - set the spill_ptr to an immediate
incsip *immediate* - increment/decrement the spill_ptr with a signed immediate

Since spills are generated and consumed by the compiler alone with no programmer intervention, the *spill space* can be considered as part of the micro-architectural state which need not be visible. This implies two things: a. spill stores do not necessarily reach memory and there is no guarantee

on the spill order for spills that do reach main memory, and b. all memory locations used for register spills are inaccessible to user code.

The simplest implementation treats the new instructions as simple stores, loads, and register operations, where the *spill pointer* (SIP) can be either a state register or a general purpose register. The only advantage of the new instructions in this scheme is that it provides memory disambiguation between regular loads and stores, and their fill/spill memory operations (using property (b)).

More sophisticated implementations can take advantage of the new information inorder to prevent spills from reaching memory, and possibly eliminate the execution of spills and fills altogether. The basic mechanism required for this approach is a *tracker* which allows the processor to identify the spill/fill location at the frontend section. This device is similar to the one proposed in [4] but simpler and more robust, since the SIP can only be modified in trackable ways.

A portion of the main memory can be allocated to store the necessary spills. A register keeps the pointer to the start of this part of memory. This register should be a special purpose state register. A general purpose register can also be used but this is going to affect the total number of general purpose registers available for the processor and certain precautions must be taken as explained in [4]. The address of the spill is calculated in the frontend using a dedicated ALU, with this register as the base and adding an immediate value provided in the spill/fill instruction. Since the compiler controls all accesses to this register using the strict syntax of the new instructions, all spill/fill references can be accurately calculated in the frontend.

A direct method for avoiding spilling to memory is to re-use the physical register that was to be spilled for the corresponding fill, based on the proposal in [5]. This is done by tagging the physical register with the current value of the SIP (the SIP is now a state register which is updated by the frontend only). When a fill arrives, its SIP is looked-up in the physical register tags. If the physical register that holds the spilled value has not been reclaimed a tag match occurs, and the execution of the spill / fill is transferred to the frontend, without consuming regular execution resources and memory bandwidth. The reclaiming of spilled physical registers can be delayed by providing two free-lists to the renamer, a regular free-list and a low priority free-list that contains spilled registers and is only used as a last resort. Once a spill is serviced (its corresponding fill_kill has been processed) the physical register is untagged and transferred to the regular free-list.

When a tagged physical register must be re-used for a new value, its current value must first be spilled to memory. This is done in several steps:

1. Stalling the renamer (which is waiting to use the register)
2. Injecting a regular store instruction of the register value to the location pointed to by the tag)
3. Restarting the renamer when a register becomes available on the regular free-list, either the injected store is executed or some other register is reclaimed.

This is illustrated by the pseudo code segment in Figure 8 that is modified from what is shown in Figure 1.

1	use	R_1	P_1	
2	use	R_2	P_2	
3	*spill*	R_1	P_1 -> Mem	Tag P_1
4	ld	R_1	Mem -> P_3	
5	use	R_2	P_4	
6	ld	R_1	Mem -> P_1	P_1 -> Mem Mem -> P_1 Untag P_1
7	*fill*	R1	Mem -> P_2	Mem -> P_2 Tag P_2

Figure 8. Example Code

In this example, line 6 is pushed down to line 7, and in its place, a load instruction is added. This greatly changes the dynamic behavior of the processor. First of all, we do not have enough physical registers to support the renaming of the ld instruction. As a result, a store must be injected to save the contents of P_1 to memory. Only after this injected store, can we load the data from memory into P_1. In line7, we actually want to fill R_1. This time, suppose we can rename the register to P_2. We then use a regular load instruction to load the data from memory to the physical register.

The most aggressive technique adds another level of register hierarchy -- a *register cache*. This register cache is not part of the register name space seen by the renaming unit. Instead, only the spill/fill controller can use it. This effectively adds more registers to spill to. Figure 6a suggests that a 64-entry register file should capture a vast majority of the outstanding spills. The register-cache is managed by the hardware, but can rely on compiler hints for which register to write to memory when is is full. A compiler/profiler can provide information as to how soon a spilled register will be consumed, thus lowering the number of spills that reach memory.

To support a register cache means must be provided for transferring a value from the cache to memory. Since all SIP addresses are virtual, spills must go through full address generation and the TLB before being sent to memory. A brute force approach is to provide a AGU unit and a cache port to the register cache. However, a more elegant solution is to inject a store when a value has to be spilled. The problem now, is which physical register to use for the store. Two simple solutions are: a. reserve at least one physical register for the sole purpose of handling actual spills, or b. buffer the current physical register that is being renamed and temporarily use it to spill the value from the cache (similar to the method described in the non-cache approach). A more complicated scheme is to employ some sort of *register stealing* as proposed in [6].

This register-cache can be used as a novel and effective way of reducing the cost of the physical register file. The number of compiler-visible registers can be reduced to the size of the lowest level of the register hierarchy. Thus the compiler will generate spill / fill instructions for moving registers into the higher hierarchies. In order to effectively manage the hierchiacal structure, compiler hints are provided as to which register should be spilled to a higher level.

5.0 Conclusion

We hypothesize that using an explicit spill and fill instructions to differentiate spilling and filling from other store and load instruction increases ILP and improves memory bandwidth. This hypothesis is proven through simulation using the SimpleScaler simulator running the SPECint95 benchmarks. With the explicit spill and fill instructions, we achieved an improvement of 22% more ILP and an reduction of 35% stores and 19% loads, on average. We also find that adding a 64-entry second level registerfile that is dedicated for spilling can capture 92-100% of all the outstanding spills, thus eliminating their memory references.

6.0 Acknowledgment

The authors would like to thank Lance Hammond for the technical conversation about SPEC95 benchmark.

7.0 References

[1] The GNU Project, “GCC Online Documentation”, http://www.gnu.org/software/gcc/onlinedocs/ .

[2] D.C. Burger and T. M. Austin, “The Simplescalar toolset, version 2.0”, Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[3] SPEC newsletter, Fairfax, Virginia, September 1995.

[4] M. Bekerman, A. Yoaz, F. Gabby, S. Jourdan, M. Kalaev, R. Ronen, “Early Load Address Resolution Via Register Tracking”, in the proceedings of 27th International Symposium on Computer Architecture, June 2000.

[5] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, A. Yoaz, “A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification”, in the proceedings of 31st International Symposium on Microarchitecture, November 1998.

[6] T. Monreal, A. Gonzalez, M Valero, J. Gonzalez, V. Vinals, “Delaying Physical Register Allocation Through Virtual-Physical Registers”, in the proceedings of 32nd International Symposium on Microarchitecture, November 1999.