McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures
and Parallel Systems

Effective Utilization of the Reorder Buffer for Short-Lived Variables

Luis A. Lozano C.
Guang R. Gao

ACAPS Technical Memo 86
July 1994

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7

Abstract

In this paper, we have observed an interesting phenomenon in superscalar architectures with aggressive hardware mechanisms (such as *register renaming* and *reorder buffers*): a significant number of program variables are "short-lived" in the sense that their whole live ranges occur entirely within the reorder buffer. Therefore, upon completion, the values of these short-lived variables do not need to be written (committed) back to the register files. Based on this observation, we have proposed a compile-time analysis method (called short-live-range analysis) and a simple architecture feature to avoid the useless commit of these short-lived variables. Furthermore, we propose a compiler mechanism to directly assign these variables to the reorder buffer (instead of the register file), thus decreasing the total requirements on the register file.

We have implemented our scheme and our simulation results show: (1) the short-live-range analysis and the proposed architecture feature can be successfully used to avoid the useless commit of instructions to the register files; (2) the above mechanism can reduce the number of write ports to the register files without affecting performance; (3) the allocation of short-lived variables to the reorder buffer can significantly reduce the introduction of spill code and improve the overall performance.

Contents

List of Figures

List of Tables

1 Introduction

Superscalar architectures exploit the instruction level parallelism found in programs by issuing and executing several independent instructions simultaneously. Examples of superscalar machines include the IBM RISC-6000 architecture [BW90], the Motorola 88110 [DA92], the Intel Pentium [AA93], the Hewlett-Packard PA7100 [AAD+93], the TFP [Hsu94] and the PowerPC [DOH94, SD94].

In a superscalar architecture, significant hardware resources are provided to support *dynamic instruction scheduling* . This enables the architecture to find multiple independent instructions to be issued and executed out-of-order to fully utilize the multiple functional units in a superscalar processor. One architecture feature designed to support out-of-order instruction issue is *register renaming* [Kel75]. Under register renaming, additional registers that are not programmer visible are dynamically allocated for each new value generated, thus avoiding the anti and output dependencies. The out-of-order processing of instructions also leads to the difficult problem of saving a consistent in-order state of the sequential execution of the program [Joh91]. Elaborate hardware mechanisms have been devised to solve this problem, including the *reorder buffer* [PGH+87, Joh91] and the *completion buffer* [SD94]. An open issue is how these aggressive hardware features can be effectively utilized (by the compiler designers and architects together).

We believe that, in order to meet such challenges, it is important that aggressive architecture features for dynamic instruction scheduling (such as register renaming and reorder buffers) not be kept entirely hidden from the compiler optimizations. Instead, the compiler writers and the architects must interact more closely. We must be ready to go beyond the traditional instruction set specification in order to explicitly utilize these new features and achieve higher performance.

In this paper, we first present an important observation for superscalar architectures with the aforementioned aggressive hardware mechanisms: a significant number of program variables are "short-lived" in the sense that their whole live ranges occur entirely within the reorder buffer. Therefore, upon completion, the values of these short-lived variables do not need to be written (committed) back to the register files. We present experimental evidence (in Section 2) which demonstrates that a significant portion — often more than 90% — of the values generated in programs have short live ranges. Under the current superscalar implementation, however, the values of these variables are unnecessarily retired (committed) to the register file upon leaving the reorder buffer. This constitutes an unnecessary runtime overhead and increases the number of hardware resources required, such as the number of register ports.

Based on this observation, we have proposed a compile-time analysis method (called short-live-range analysis) and a simple architecture feature to avoid the useless commit of these short-lived variables. Moreover, we propose a novel register allocation scheme by which a compiler can directly assign these variables to the reorder buffer (not to the registers), thus decreasing the total requirement to the register file.

We have implemented our scheme on McCAT – The McGill Compiler-Architecture Testbed – which includes an optimizing C compiler [HDE+92] and a cycle-by-cycle superscalar simulator [Mou93]. Our simulation results have demonstrated that:

- The proposed short-live-range analysis can successfully capture most of the short-lived variables: on the average between 80% and 92% of the variables are detected to be short-lived. Furthermore, the combined architecture and compiler method can effectively make use of such analysis and eliminate a great majority of useless writes to the register files: on the average between, approximately, 77% and 89%.
- The mechanism devised to avoid the useless commits of instructions can be used to reduce the number of write ports to the register files without affecting performance. Only a 1% performance loss is detected after reducing the number of write ports to the register file from 4 to 1.
- The proposed method of allocation of short-lived variables to the reorder buffer can reduce the number of registers needed and, consequently, the amount of spill code introduced by the register allocator thus substantially improving the overall performance. For instance, the average improvement when the register pressure is high, is quite significant: between, approximately, 16% and 35%.

We have structured this report as follows: in Section 2, we discuss the background of superscalar architectures. We also discuss the generic superscalar architecture model and we present an example to illustrate the problems studied. These problems are formulated in Section 3. Section 4 describes our general strategy to solve the stated problems. Sections 5 and 6 provide a detailed solution to the problems. In Section 7, we present our experimental results, and in Section 8, we discuss the related work. Finally, Section 9 contains our conclusions.

2 Background and Motivation

In Section 2.1, we present background information on the superscalar architecture model to be used in the rest of this paper. In Section 2.2, we present a motivating example to highlight the problems to be studied.

2.1 The Superscalar Architecture Model

Figure 1 shows the structure of a generic superscalar architecture model [Joh91, Lai92, Mou93] which will be adapted in this paper. The superscalar architecture model has two important

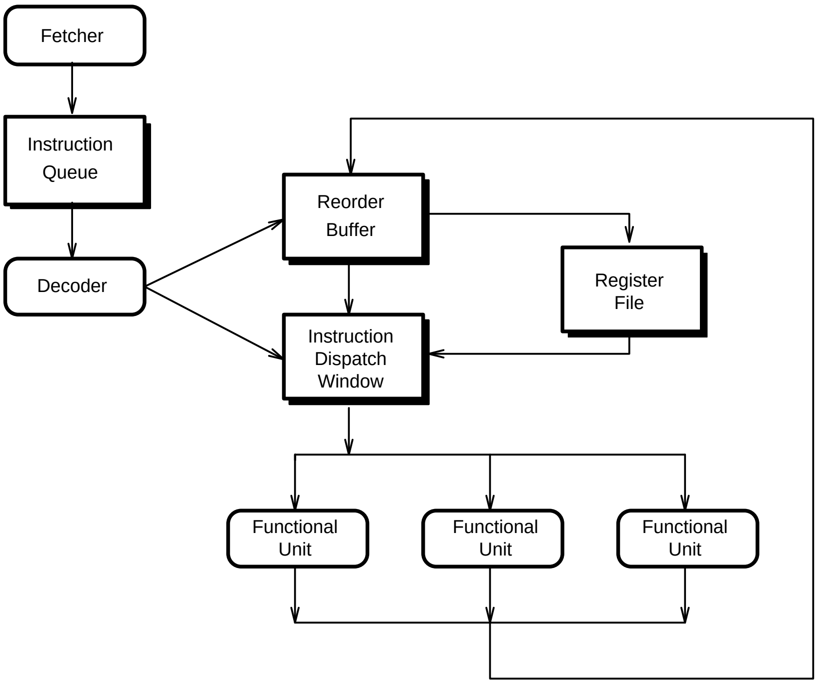


Figure 1: A generic Superscalar Architecture

components: the instruction (dispatch) window and the reorder buffer. The instruction window serves as a pool of instructions from where the instructions that are ready to execute are issued to the functional units. The reorder buffer is a FIFO structure which ensures that instructions (upon leaving the buffer) modify the register file in program order and provides the mechanism to support speculative execution and precise interrupts.

Let us briefly overview how the elements in this model interact to execute instructions concurrently. Instructions are brought from memory by the fetcher which aligns and merges them before putting them in the instruction queue. The decoder reads several instructions each cycle from the instruction queue and puts them in the reorder buffer and in the instruction window. Instructions enter the reorder buffer in program order, and are assigned a tag which identifies them and allows automatic renaming of the destination register. Register renaming is performed to improve the performance of the model because it avoids output and anti-dependencies which can limit the parallelism of instruction execution. At the same time when an instruction, say s, enters the instruction window, its operands are searched for in the reorder buffer. If an operand is found in the reorder buffer and its value has already been calculated, the value is directly used as the operand for s in the instruction window. If the operand is found in the reorder buffer but its value has not been calculated, then the tag (corresponding to the instruction which calculates the value) is used as the pending operand for s in the instruction window. Finally, if the operand is not encountered in the reorder buffer then its value is obtained from the register file (instead of the reorder buffer).

The issue stage selects from the instruction window the instructions whose operands have already been calculated, checks the availability of the functional units and sends them to be executed in the functional units. The instructions are, therefore, *dynamically scheduled* according to the availability of the operands and the functional units. This mechanism thus allows the *out-of-order issue* of the instructions to the functional units. Once an instruction has been executed (completed by a functional unit), its resulting value is written back to the reorder buffer and forwarded to the instructions whose pending operand tags correspond to the tag of the completed instruction. It is important to note that values computed by the functional units are not written back to the register file directly; they are written back to the reorder buffer which allows the *out-of-order completion* of instructions. Instructions can complete and have their values written to the reorder buffer in any order. It is at the commit stage when instructions at the head of the reorder buffer are retired to the register file. This FIFO structure of the reorder buffer also provides the undo capability necessary to support speculative execution. When a branch is executed and its prediction turns out to be incorrect, all the instructions that come after the branch are marked so they can be discarded later at commit time.

The basic principle demonstrated by the above model has been applied to different superscalar architectures with some variations. For example, alternatives to the centralized instruction window include decentralized windows called *reservation stations* [Tom67], where there is one window of operations for each functional unit. The reservation stations also allow out-of-order issue of the instructions in different stations and within the same station.

2.2 Useless Commits: A Motivating Example

One important observation of this paper is that a significant portion of the values generated by the functional units are used only while they reside inside the reorder buffer. This is because a large percentage of the values generated have a very short live range (mainly because of the large amount of temporary variables introduced by the compiler).

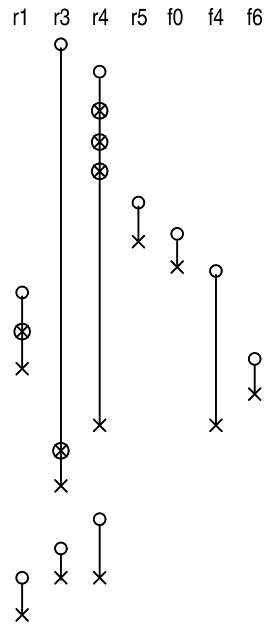
As an example, consider the small loop extracted from the Tomcat benchmark and reproduced in Figure 2(a). Figure 2(b) shows the DLX [HP90] assembly code (generated with the DLX–GCC compiler) for the body of the loop. Figure 2(c) shows the live ranges for the different values generated and the respective register assignments. In this figure, small circles ($\circ$) denote definition points while cross signs ($\times$) denote the points where the values die. Overlapped cross and circle signs ($\otimes$) denote redefinition points. In this example, it can be seen that for all live ranges, once a new value is defined, it is last used a few instructions later. If we count the length of the live range by the number of instructions between the producer and the last consumer, we can see that the length of the longest live range is 14 instructions. If this code were to be run on a machine with a reorder buffer of 16 entries, all instruction commits could be useless because all values produced could be consumed while they reside in the reorder buffer. This makes the commit of such short-lived values to the register files useless!

```
for (I = 1; I <= N; I++) {
  X[I][1] = ((double) (I-1))
          /(double) (N-1);
}
```

(a) C Program

```
L5:
  lw r3,0(r6)
  slli r4,r3,#2
  add r4,r4,r3
  slli r4,r4,#3
  add r4,r7,r4
  add r5,r3,#-1
  movi2fp f0,r5
  cvti2d f4,f0
  lhi r1,(LC0>>16)&0xffff
  addui r1,r1,(LC0&0xffff)
  ld f6,0(r1)
  divd f4,f4,f6
  sd 8(r4),f4
  add r3,r3,#1
  sw 0(r6),r3
  lw r4,0(r6)
  addi r3,r0,#4
  sle r1,r4,r3
  bnez r1,L5
```

(b) Assembly code



(c) Live Ranges

Figure 2: Examples of short-live-range variables

Another important point to note from this example is that nine physical registers have been used for the allocation of the temporal variables used in the body of the loop. This is a waste of resources because, as we pointed out before, the values stored in these registers are never being directly used from the register file. A natural question is: should they have been allocated to the physical (architected) registers in the first place?

2.3 Experimental Observations

After considering this example, we want to determine how often these useless commits occur during the execution of real programs. For this purpose, we modified our superscalar simulation testbed to count the percentage of useless commits for a set of benchmarks. The results of these experiments are summarized in Table 1. A description of the testbed and the benchmarks will be given in Section 7.

From the results presented in this table, it can be seen that a large percentage of the instructions committed to the register file are useless. The data is presented for different sizes of the reorder buffer. The percentage of useless commits increases with the size of the reorder buffer because there is a bigger chance that the last use of a value and its definition point reside in the reorder buffer at the same time. Even with a reorder buffer of 8 entries the percentage of useless commits is very high (on average 89.98%). With a reorder buffer of 32

Benchmark	Buffer Size		
	8	16	32
Bubble	93.82	95.93	97.14
L8	89.66	98.81	98.81
L14	91.65	95.93	97.08
L8unroll	89.54	98.70	98.70
L14unroll	91.77	95.87	97.12
Linpack	92.32	92.61	93.33
Quickrand	81.81	87.52	88.19
Tomcat	95.95	97.61	97.78
Whetstone	83.29	90.21	93.04
Average	89.98	94.80	95.69

Table 1: Percentage of Useless Commits

entries the percentages are even higher, on average 95.69%. These results suggest that all these useless commits constitute a waste of resources and leave space for architecture and compiler optimizations.

3 Problem Formulation

In this section we formulate the problems to be studied in this paper.

Let $d_0, d_1, d_2 \ldots d_n$ be the definition points for a set of values $v_0, v_1, v_2 \ldots v_n$. Let $u_0, u_1, u_2 \ldots u_n$ be the corresponding last-use points for this set of values; We define the live range r_i of a value v_i as the sequence of instructions in the interval $[d_i, u_i]$ and the length of r_i, $\mathcal{L}(r_i)$, as the number of machine instructions between d_i and u_i including d_i and u_i. Assume we have a superscalar machine $\mathcal{M}$, like the one described in Section 2.1, with a reorder buffer of length $\mathcal{LR}$ entries. We say that the instruction associated to the definition point d_i of v_i is a *useless commit* to the register file if all the references to v_i are not obtained from the register file but directly from the reorder buffer. Note that for an instruction at d_j to be a useless commit, it is necessary that $\mathcal{L}(r_j) \leq \mathcal{LR}$. We say a live range r_i is short if d_i is a useless commit. Consequently, we call a variable x a *short-lived variable* if all the live ranges associated to x are short.

The problems to be studied in this paper are as follows:

- **Problem 1 - Reduction of Useless Commits:** Given a machine $\mathcal{M}$ with a reorder buffer of $\mathcal{LR}$ entries and a program $\mathcal{P}$, we want to develop architecture mechanisms and compiler techniques to reduce the number of useless commits (due to short-lived variables) in $\mathcal{M}$, while executing $\mathcal{P}$.

- **Problem 2 - Allocation of short-lived variables:** In conjunction with the solution for Problem 1, we want to propose a scheme which ensures that the short-lived variables in $\mathcal{P}$ do not occupy physical (architected) registers of the machine whenever possible.

Solving Problem 1 will allow the architecture to reduce the number of write ports to the register file without affecting the execution time of the program. The reduction of write ports is an important issue because it simplifies the structure of the data path of the processor [UJ92, Joh91]. The complexity of implementation of the reorder buffer, the register file and the associated busses depends on the number of ports to the register file. The area complexity of a register file is roughly proportional to the square of the number of ports [CDN92]. Due to the complexity of the register file, the cycle length for many architectures is determined by the access time to the register file. Recent designs like the PowerPC 604 [SD94] or the TFP [Hsu94] provide register files with more than 10 ports in the register file. Even 10 register ports may not be enough, and the designers added a dedicated completion stage to its pipeline to handle the possible contention [SD94].

Solving Problem 2 will allow the compiler to utilize more efficiently the physical (architected) registers by using them exclusively for the allocation of variables with long live ranges. Variables with short live ranges will be allocated to the reorder buffer using an mechanism to be defined in Section 6. As a result, the number of available physical registers for long live ranges will effectively increase and, therefore, register spilling will be reduced. In other words, the reorder buffer will be seen and used by the compiler as an "extension" of the programmable registers during the register allocation process.

4 Solution Strategy

In this section, we briefly outline the issues and the overall solution strategy to the problems stated in the previous section. A detailed discussion of the solutions proposed will be presented in Sections 5 and 6.

As observed in the previous section, a large percentage of the instruction commits are useless. Therefore, we propose to reduce this percentage (Problem 1) by providing the architecture with specific information, so that at commit time the architecture can decide whether a value should be committed to the register file or just discarded. This information should be collected by the compiler, by analyzing the characteristics of the live ranges of the variables, and provided to the architecture through the instruction set. Thus, it is the responsibility of the compiler to mark the instructions that are going to be discarded at commit time.

To solve Problem 2 we propose to modify the register allocation process, so that variables whose live ranges are not committed to the register file are not assigned to physical registers since these live ranges are not going to make use of them. We maintain that these variables should be assigned to and stored in the reorder buffer instead of the register file. Then, the reorder buffer will be considered an extension of the register file in which the values of short live ranges will be temporarily stored before they are discarded at commit time. The actual storage for the value will be dynamically allocated by the renaming mechanism of the architecture.

However, in order to be able to apply these solutions, it is necessary to address the following two issues. The first issue is that, with the architecture mechanisms described for the generic superscalar model in Section 2.1, the definition point of a value is committed to the register file as soon as it reaches the head of the reorder buffer, without waiting for the last use to enter the reorder buffer. As a consequence, the compiler cannot safely conclude that the definition point of a value will be a useless commit based only on the fact that the live range associated has a length smaller than the size of the reorder buffer. To address this issue, the compiler will mark the appropriate definition and last use points of the value in the code, while the architecture will be augmented to ensure that the instruction defining the value will not leave the reorder buffer before the last use enters the reorder buffer. This solution strategy will be further elaborated in Section 5 where the solution schemes are presented.

The second issue is related to the speculative execution of instructions. The problem of knowing at compile time when an instruction is going to be a useless commit is even more difficult in the presence of speculative execution. After a definition point has entered the reorder buffer, one or several branches can be predicted before the last use of the value enters the reorder buffer. Yet, even after the last use has entered, it is not safe to say that the definition point can be discarded once it arrives to the head of the reorder buffer because some of the branches in the middle could have been wrongly predicted. If the definition point is discarded and the prediction for one of the branches happens to be incorrect, then another use of the value can be fetched from the new path, and this use will not be able to find the value in the reorder buffer nor in the register file. To solve this problem the architecture would have to put more restrictions on the instruction commit process: an instruction at the head of the reorder buffer could not be discarded until the last value had entered the reorder buffer and there were no unresolved branches between the definition point and the last use of the value. Such a mechanism would be difficult to design and implement. Consequently, to address this issue we based our strategy on the following observation: there is a high probability that for each short live range its definition and last use points are found in the same basic block. To verify this observation, we performed a simulation study and the results of our measurements for different reorder buffer sizes are presented in Table 2.

Comparing these results with the ones presented in Table 1, it can be seen that most of the short live ranges do not cross the basic block boundaries. Even for a reorder buffer of 32 entries, the difference of the measurements presented in Tables 1 and 2 is at most 4%. The

Benchmark	Buffer Size		
	8	16	32
Bubble	93.83	95.90	95.92
L8	89.66	98.81	98.81
L14	91.26	95.16	95.16
L8unroll	89.54	98.70	98.70
L14unroll	91.39	94.85	94.85
Linpack	92.31	92.50	92.64
Quickrand	81.37	84.11	84.50
Tomcat	96.31	97.45	97.57
Whetstone	82.82	89.26	89.26
Average	89.83	94.08	94.16

Table 2: Percentage of Useless Commits at the basic block level

differences are further reduced when we consider the measurements for reorder buffers of 8 and 16 entries. This small difference confirms that our mechanism can be used, without a significant loss of precision, to detect the short live ranges that do not cross the basic block boundaries, thus avoiding in a simple way the problem posed by the speculative execution of instructions. In this way, we are applying the principle of making the common case fast [HP90] by avoiding the use of a complicated hardware mechanism that would only be necessary in a small percentage of the cases. According to this strategy, the short-live-range variables whose live ranges cross the basic block boundaries will be considered as not short-lived variables in our study.

As it can be seen, our solution strategy for the stated problems requires a combined scheme in which the architecture provides some support for the compiler analyses, and the compiler gives information to the architecture so that some decisions can be taken at run time. This scheme will be further elaborated in the next two sections.

5 Reducing the Number of Useless Commits

In this section, we present a detailed description of the architecture and compiler mechanisms required to support the solution strategy, proposed in the last section, for the problem of reduction of useless commits (Problem 1). We first present the possible architectural modifications followed by the compiler analysis used.

5.1 Architectural Support

In order to allow the compiler to decide whether the definition point of a live range will be a useless commit (and if so, to tag it), the architecture must provide a mechanism to ensure that the whole live range of a tagged value will occur inside the reorder buffer. Therefore, the architecture has to impose more constraints on the commit process of an instruction to guarantee that the definition point of a value chosen by the compiler will not be discarded until the last use enters the reorder buffer. This can be implemented in two ways:

- **Scheme 1: Marking the last-use of a value**

  The definition point of a value is not committed before the last use enters the reorder buffer. The compiler is responsible for informing the architecture which of all the uses of the value is the last one. When the last use enters the reorder buffer an associative search is performed in the reorder buffer and the definition point is marked so that when it arrives at the head of the reorder buffer it can be discarded. This should not increase execution time since the architecture in the base model is already performing the associative search to get the value of the operands.

  This mechanism implies a modification in the instruction format as illustrated by the following example:

```
Inst. A    add &r3, r4,r5  ; Definition Point
           ...
Inst. B    sub r5, r4,r3   ; Use of r3
           ...
Inst. C    sw  0(r5),&r3   ; Last use of r3
```

  In this example, the ampersand sign is used to inform the architecture that the definition of r3, in Inst. A, can be discarded at commit time and to flag the last use of r3, in Inst. C. Thus, according to this mechanism, the (untagged) value assigned to r5, in Inst. B, will be committed to the register file as soon as it reaches the head of the reorder buffer. However, two conditions must be met in order for the value of r3, in Inst. A, to be scrapped: Inst. A must have reached the head of the reorder buffer and Inst. C must have entered the reorder buffer.

  Special care has to be taken by the compiler so that the use of this mechanism does not produce stalls in the decode stage. If the length of the reorder buffer is $\mathcal{LR}$ and the decode bandwidth of the processor is $\mathcal{DB}$, then the length of the live ranges chosen by the compiler must not exceed $\mathcal{LR} - \mathcal{DB}$ to avoid the case where instructions cannot enter the reorder buffer because another instruction is at the head waiting for the last use to enter the reorder buffer and stopping the commit of other instructions. So, as an example, if $\mathcal{LR}$ is 16 and $\mathcal{DB}$ is 4 then the length of the live ranges chosen to be discarded should not exceed 12 instructions.

- **Scheme 2: Keeping a minimum number of instructions in the reorder buffer**

  In this mechanism, an instruction cannot be committed unless there is a minimum number of instructions in the reorder buffer. Here again, special care has to be taken when choosing the minimum number of instructions to be kept in the reorder buffer in order to avoid decode stalls. The minimum number of instructions $\mathcal{MLR}$ will be, therefore, an architectural parameter which depends on the reorder buffer size $\mathcal{LR}$ and the decode bandwidth . Knowing $\mathcal{MLR}$, the compiler will be able to decide which live ranges will completely occur inside the reorder buffer and mark them, so the architecture can safely discard the definition point at commit time.

  An advantage of this mechanism is that the compiler does not need to flag the last use of a value to be discarded. Only the definition point has to be flagged. Hence, our previous example for the change in the instruction format will look like the following under the use of this mechanism:

```
Inst. A     add &r3, r4,r5  ; Definition Point
            ...
Inst. B     sub r5, r4,r3   ; Use of r3
            ...
Inst. C     sw  0(r5),r3    ; Last use of r3
```

  The ampersand is used here to indicate that the value generated for r3, in Inst. A, can be discarded at commit time. The value will not be scrapped until the number of instructions in the reorder buffer is the minimum specified. On the other hand, the value for r5, in Inst. B, will be committed to the register file once there are enough instructions in the reorder buffer.

These two schemes are equivalent in terms of effectiveness. The selection of the mechanism will depend, then, on the complexity of the implementation. As mentioned before, Scheme 2 has the advantage that only the definition points need to be marked by the compiler. This implies a smaller change in the instruction format and, depending on the instruction set, could imply that object code compatibility could be preserved. On the other hand, the implementation of Scheme 2 could be more difficult for some architectures particularly in the presence of branch prediction. When a branch is wrongly predicted, some architectures scrap the instructions following the branch while others just mark them so that at commit time they can be discarded. For the architectures that actually scrap the instructions, it could be more difficult to keep an account of how many instructions are in the reorder buffer. In this case, Scheme 1 could be easier to use.

5.2 Compiler Mechanism

Once we have designed the architecture mechanism that allows the compiler to predict the useless commit of the instructions, we have to design the compiler analyses to decide which live ranges will be discarded and flag them so the architecture can act accordingly.

This analysis is called *short-live-range analysis*. Under our implementation, it takes as input the low level abstract syntax tree representation of the program and the set of variables used at this level, and produces as output the tree annotated with the information of which variables' live ranges can be discarded. The code generator can then take this annotated tree, and produce the instructions with the appropriate flags for the architecture.

This is a very simple analysis, very similar to live variable analysis. It does a backwards walk of the tree keeping track of the last uses and definition points of the variables. For each definition point, it calculates the distance in number of instructions to the corresponding last use. For the variables having more than one live range, the analysis takes the maximum of the lengths. For the case of variables whose live ranges cross the basic block boundaries, the analysis assumes that the length of the range is larger than the length of the reorder buffer thus considering these variables as not short-lived variables. Here, we remind the reader about the observation made in Section 4: the majority of the short-lived variables do not span across basic blocks. After the analysis is performed we obtain, for all the variables, the length of the longest live range. With this information and the knowledge of the length of the reorder buffer and the decode bandwidth, the analysis can decide which definition points can be scrapped at run time.

6 Allocating Short-Lived Variables

In this section we present in more detail the description of the proposed mechanism for the allocation of the short-lived variables (Problem 2). We only present the compiler side of the mechanism since the required architecture modifications are as described in the last section.

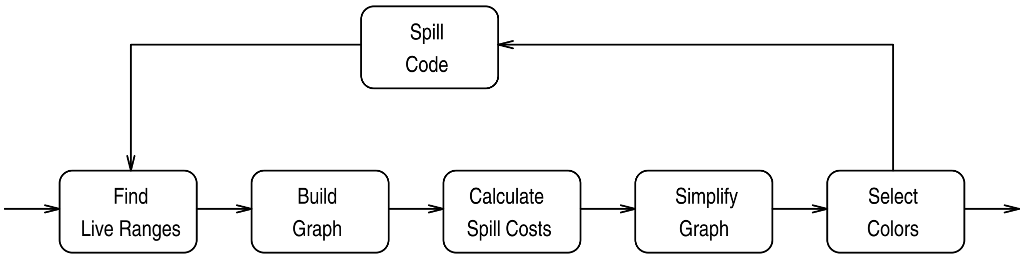
Even if the previously described architecture mechanism is used, the short-lived variables are still being allocated to physical registers although they are no longer going to reside in the register file. This is a waste of precious register resources which can be better utilized to obtain an increase in the performance of programs with high register pressure. Hence, our idea is to expose the reorder buffer to the compiler as an extension of the programmable registers for storing short-lived values.

Under our proposal, the space in the reorder buffer is treated as additional registers which we call *symbolic registers*. There are as many symbolic registers as entries in the reorder buffer. However, the allocation of a symbolic register to a specific entry in the reorder buffer is not fixed statically at compile time. The actual association between a symbolic register and a reorder buffer entry is done at runtime when the associated instruction enters the reorder buffer and the renaming mechanism renames the symbolic register.

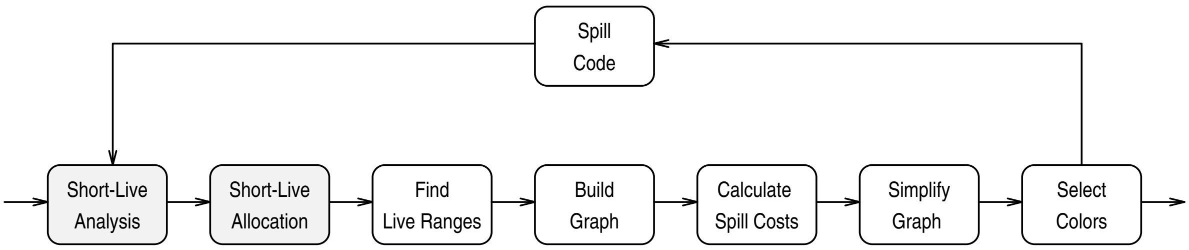
The goal of the compiler is, then, to assign the short-lived variables to the symbolic registers and allocate the remaining variables to the physical (architected) registers. To achieve this, we need to modify the register allocation process so that short-lived variables are allocated separately. As a consequence, we modified our register allocator to perform the following steps:

1. Short-live-range analysis
2. Allocation of short-lived variables
3. Modified Chaitin-like allocation for remaining variables
4. Introduction of spill code

These steps are depicted in Figure 3(b) where they can also be compared to the steps in the traditional register allocation process which are represented in Figure 3(a).



(a) Traditional Register Allocation



(b) Register Allocation with Short-Live variables analysis

Figure 3: Traditional Register Allocation vs. Short-Live variable allocation

First, the compiler performs the short-live-range analysis, as described in the previous section, to find out which variables can be allocated to the symbolic registers. Now, since the live ranges of these variables do not cross the basic block boundaries, they can be allocated in linear time using a very simple algorithm for basic block register allocation [ASU88]. Using this algorithm, independent live ranges that belong to the same variable can be allocated to different registers. It is important to note here that, since the number of symbolic registers available is equal to the number of entries in the reorder buffer, and since the length of the live ranges does not exceed the length of the reorder buffer (the short-live-range analysis ensures this is true), the number of symbolic registers is always enough to allocate all the short-lived variables without requiring the introduction of spill code.

After all the short-lived variables are allocated, the next step is to allocate the remaining variables using Chaitin's allocator with the Briggs improvement [CAC+81, Bri92]. Since many of the variables have already been allocated by the basic block allocator, in this step the size of the interference graph is smaller than what it would be for the traditional allocator, thus simplifying the problem. This also implies that fewer variables are competing for the physical registers which means a decrease in the amount of spill code required.

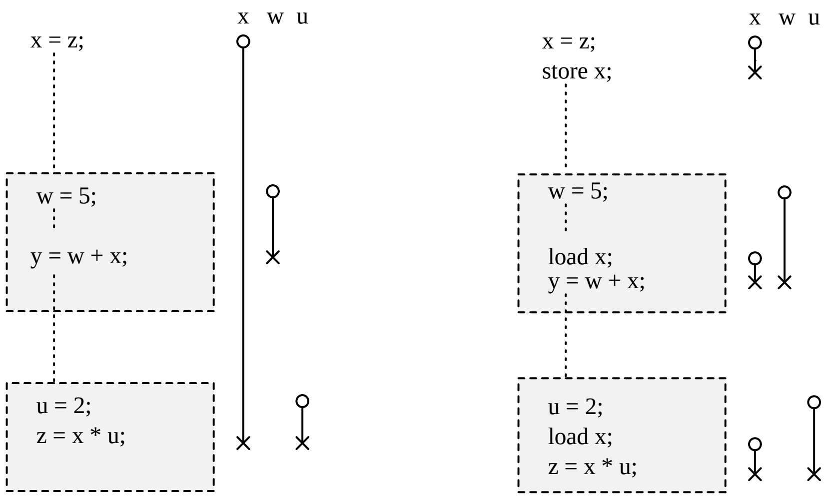
After the graph is simplified and the variables colored, some spill code might be required. The introduction of spill code has two effects on our mechanism: first, the variables that are spilled now become short-lived variables; second, because of the introduction of loads and stores for the spill code, some variables that were considered short-lived variables before are not anymore. These two effects can be better understood by examining the example presented in Figure 4. Figure 4(a) presents a fragment of a program together with the representation of the respective live ranges before the spill code is introduced. The gray boxes represent zones where the register pressure is assumed to be high enough to require the introduction of spill code[1]. In the example, x is not short-lived and has several uses, while w and u are short-lived variables. For the purposes of our example, we are going to assume that the length of the live range of w is just on the limit of being considered a short-lived variable. Let us assume that, since the register pressure is high, the register allocator selects x to be spilled, and let us examine the effect of introducing spill code for this variable. Figure 4(b) presents this effect. The first effect is that since x was spilled, its live range is now composed of several small live ranges and x becomes now a short-lived variable that can be allocated using symbolic registers. This is an advantage of our scheme over Chaitin's allocator where once the variables are spilled, the whole process has to be repeated again because spilled variables continue to interfere with the other variables in the interference graph. The second effect is that since the length of the live range for w was on the limit of being considered short, and since some instructions were introduced for the spill code of x, then the live range for w is not short anymore and w has to be allocated using the physical registers. This forces the repetition of the whole process to check if anymore spill code is required for the variables that interfere with w. We have observed, though, that this does not happen too often and that our process converges faster than in the traditional Chaitin allocation. Finally, note that despite the fact that the live range for u is longer now, u is still a short-lived variable and can still be allocated using symbolic registers. This implies that no more spill code is going to be introduced in this zone with high register pressure.

7 Experiments and Results

In this section we present our experimental results and their analysis.

All of our analyses and experiments were carried out using the McGill Compiler and Architecture Testbed (McCAT) testbed. This testbed consists of an optimizing C compiler

[1]We assume that some registers are occupied by other program variables not shown in the figure.



(a) Before spill code

(b) After spill code

Figure 4: Effect of spill code on the allocation of short-lived variables

[HDE+92] and a superscalar architecture simulator (SuperDLX). SuperDLX simulates many of the features described by Johnson [Joh91] including dynamic scheduling, branch prediction, register renaming and the use of a reorder buffer. A description of SuperDLX can be found in C. Moura's Master's thesis [Mou93]. We ran our experiments on 9 benchmark programs. The benchmarks (see Table 7) consist of 2 Livermore loops (both rolled and unrolled versions), two integer benchmarks (Bubble and Quickrand), one scientific kernel (Linpack), one floating point benchmark from SPEC (Tomcat) and one synthetic benchmark (Whetstone).

Benchmark	Description
Bubble	Recursive bubble sort
L8	Loop 8 of the Livermoore loops
L14	Loop 14 of the Livermoore loops
L8unroll	Loop 8 unrolled twice
L14unroll	Loop 14 unrolled twice
Linpack	Kernel routines from the "LINPACK" linear algebra package
Quickrand	Recursive quicksort
Tomcat	Mesh generation
Whetstone	Floating point intensive synthetic benchmark

Table 3: Benchmarks

To show the effectiveness of the solution methods proposed, we performed the following measurements:

1. Effectiveness of the short-live-range variables analysis in terms of the percentage of program variables that are detected to be short-live-range variables and in terms of the percentage of writes to the register files that can be discarded by using this analysis.

2. Effect of the reduction of the write ports to the register files on the performance of the base model and on the performance of our proposed (optimized) model.

3. Effectiveness of the allocation of short-lived variables to the symbolic registers allocated in the reorder buffer in terms of the percentage of reduction in the execution time for the different benchmarks.

All of the above measurements were performed by simulating the behavior of a *base model* of a superscalar architecture with the characteristics described in Table 4. Some of the parameters of the base model, like the decode bandwidth, the number of memory accesses per cycle and the memory latency, were selected according to real values found in some recent aggressive superscalar architecture designs. The base model uses a reorder buffer size of 16 entries and assumes 4 write ports per register file. According to the different measurements, we vary some parameters of the base model, e.g.: the reorder buffer size between 8, 16 and 32 entries (see Section 7.1) and the number of register ports per register file (see Section 7.2). In such cases, the sizes of the instruction dispatch window and the load and store buffers are always kept equal to the size of the reorder buffer. For simplicity, we assume that the base model and its variations have a perfect cache and we briefly comment on the effect of a non-perfect cache on our results.

Instruction queue	16 instructions
Decode bandwidth	4 instructions per cycle
Memory latency	2 cycles
Memory accesses	1 per cycle
Address resolution functional units	1
Other functional units	5 of each type
Latency of other functional units	1 cycle

Table 4: The Base Model of the Simulated Superscalar Architecture

The major results of our experiments are:

- The short-live-range analysis can be successfully used to avoid the useless commit of instructions to the register files. The proposed analysis successfully captures most of the short-lived variables: on the average, close to 90% (89.88) of all variables are detected to be short-lived for the base model, and more than 90% (92.53) when the reorder buffer size is increased to 32. The combined architecture and compiler method can effectively make use of such an analysis and eliminate a great majority of the useless writes to the register files: on the average 87% (87.96) for the base model and close to 90% (89.84) when reorder buffer size is increased to 32.

- The mechanism devised to avoid the useless commits of instructions can be used to reduce the number of write ports to the register files without affecting performance.
- The proposed method for allocation of short-lived variables to symbolic registers can reduce the number of registers needed and improve the overall performance. The average improvement, when the register pressure is high (only 4 registers are effectively available) is quite significant: over 30% (30.41%) for reorder buffer of size 16 and up to 35.04 when reorder buffer size is increased to 32.

We will further elaborate about these results in the following three subsections.

7.1 The Effect of Short-Live-Range Analysis and Architecture Support for Useless Commit Elimination

The effectiveness of the short-live-range analysis (explained in Section 5) is illustrated in Table 5. In this table we measured the effectiveness of our analysis for different sizes of the reorder buffer. For each size we present the percentage of variables that were detected by the compiler to be short-lived variables, and the percentage of writes that are discarded at run time. It can be seen, that even for a small reorder buffer (of size 8), on the average more than 80% of the variables used at the low level representation of the program are being detected as short-lived variables. This is caused, as explained before, by the large number of temporal variables introduced by the compiler. Most of these variables, plus the ones that are spilled to memory by the register allocator, are successfully captured by our compiler analysis: on the average close to 90% (89.88) for the base model, and 80.66% and 92.53% when the reorder buffer sizes are 8 and 32 respectively.

Furthermore, the combined architecture and compiler method proposed in this paper can effectively make use of such an analysis and eliminate a great majority of useless writes to the register files. The reduction on the average is 87% (87.96) for the base model, and is 77.12% and 89.84 % when reorder buffer size are 8 and 32 respectively.

Contrasting the results of this table against the results presented in Table 1, we can see that our analysis is able to detect a great majority of the useless commits. For reorder buffers of size 16 and 32 the difference is smaller than 8%. For a reorder buffer with 8 entries, the difference is less than 13%.

Benchmark	Reorder Buffer					
	8		16		32	
	Short live Variables	Discarded Writes	Short live Variables	Discarded Writes	Short live Variables	Discarded Writes
Bubble	81.25	79.56	89.58	89.76	91.67	91.81
L8	73.15	72.04	89.49	96.06	94.16	98.80
L14	83.33	83.95	88.33	88.37	91.67	90.78
L8unroll	81.37	70.90	95.24	95.60	96.70	98.34
L14unroll	89.35	83.97	92.73	88.41	94.29	90.80
Linpack	80.20	84.37	89.93	90.88	94.14	91.46
Quickrand	76.60	71.73	80.85	76.61	80.85	76.61
Tomcat	79.05	80.26	91.38	88.17	94.80	89.99
Whetstone	81.62	67.27	91.39	77.76	94.49	80.00
Average	80.66	77.12	89.88	87.96	92.53	89.84

Table 5: Effectiveness of the short-live-range analysis and the architecture support for reducing useless commits

7.2 The Effect of Reducing the Number of Register Ports Needed

Since the percentage of commits that can be discarded is large, we decided to measure the loss of performance of the proposed (optimized) architecture when the number of write ports per register file is restricted, and compare it to the loss of performance for the base model. The left side of Table 6 shows the relative performance obtained by varying the number of register write ports when the compiler and architecture optimization proposed in this paper is not applied. We report the results of variations of the base model when the number of write ports to the register files is restricted to 1, 2 or 3 ports per register file. From the table, it can be seen that restrictions on the number of ports to the register file can seriously affect the performance of the base model. Using, for example, only one port per register file can degrade the performance by 50%! If we increase the number of ports to two per register file we still get a performance loss of 16%!

We applied the optimization proposed to the base model and performed the same measurements. The results, in the right side of the Table 6, show that the loss in performance is, as it was expected, very small. Around 1% when using only one write port to each register file.

Benchmark	Base Model Write Ports 1	2	3	Optimized Write Ports 1	2	3
Bubble	0.42	0.78	0.92	1.00	1.00	1.00
L8	0.34	0.69	0.97	1.00	1.00	1.00
L14	0.53	0.92	0.97	0.97	0.98	0.98
L8unroll	0.34	0.68	0.97	1.00	1.00	1.00
L14unroll	0.59	0.97	1.00	0.97	0.98	0.98
Linpack	0.45	0.86	1.00	1.00	1.00	1.00
Quickrand	0.67	0.95	0.98	1.00	1.00	1.00
Tomcat	0.47	0.80	0.97	0.99	1.00	1.00
Whetstone	0.65	0.95	0.99	1.00	1.00	1.00
Average	0.50	0.84	0.98	0.99	1.00	1.00

Table 6: Performance effect of the number of write ports on the base and optimized models

7.3 Effect of the Allocation of Short-Live-Range Variables

Table 7 shows the improvement obtained in the overall execution time of the benchmarks by using the short-lived variables allocation method explained in Section 6. To show the effectiveness of the method we vary the number of registers available (before the allocation) for the register allocation process (4, 8 and 16) and the size of the reorder buffer (8, 16 and 32) from the base model. The performance improvement is calculated with the formula $(cycles(old) - cycles(new))/cycles(old)$. Where old refers to the Chaitin like allocation method and new refers to our proposed mechanism.

It can be seen from the table that the higher the register pressure, the higher the improvement obtained by our method. When the register pressure is high (only 4 registers are effectively available), the improvement is quite significant: over 30% (30.41%) for a reorder buffer of size 16, and 16.26% and 35.04 when the reorder buffer size is 8 and 32 respectively. As explained before in Section 6, this improvement is obtained because the use of symbolic registers increases the number of registers available for the allocation process, and because the spill code produced is now allocated to the symbolic registers thus reducing the possibilities of introduction of more spill code.

When the register pressure is not high, there is still some improvement obtained by our method (L14, Quickrand and Bubble allocated with 16 registers). Because of the use of the symbolic registers, routines can be allocated with a smaller number of physical registers which reduces the cost of procedure calls.

It is important to note that we are taking all of our measurements assuming a perfect cache and that we are making use of large load/store buffers. Without the use of these features

	Number of registers								
	4			8			16		
	Reorder Buffer			Reorder Buffer			Reorder Buffer		
Benchmark	8	16	32	8	16	32	8	16	32
Bubble	12.44	34.72	47.16	0.04	0.03	0.08	0.04	0.03	0.08
L8	19.75	36.35	45.89	7.27	25.86	36.98	0.03	0.05	0.07
L14	12.13	30.01	33.71	4.07	9.76	8.30	2.10	1.95	0.18
L8unroll	20.09	40.59	46.37	5.98	20.90	32.48	0.49	0.96	2.00
L14unroll	21.70	34.06	36.14	7.47	16.28	16.68	8.75	12.15	12.37
Linpack	21.82	37.94	40.06	0.28	1.40	1.15	0.70	0.94	1.67
Quickrand	17.14	29.40	34.02	0.23	2.47	5.15	0.00	1.12	3.72
Tomcat	10.39	14.43	16.45	2.56	20.83	25.53	1.35	3.46	3.29
Whetstone	10.83	16.16	15.58	4.58	10.29	10.47	6.81	13.04	12.58
Average	16.26	30.41	35.04	3.61	11.98	15.20	2.25	3.74	4.00

Table 7: Percentage of improvement obtained by allocation of short-lived variables

the improvements obtained by our method would be even greater. That is, the results reported in this section are somewhat conservative and can be used as a safe estimate of the potential improvement gained from our method.

8 Related Work

Several authors have noticed the occurrence of the useless commit of instructions to the register file. The first reference we found is in [PS88] where the authors mention the fact that the value of a register should not be updated if there is a succeeding instruction in the RUU (Register Update Unit) that will update the value of the register. They briefly discuss the complexity of the implementation of the result bus and suggest that many of the read ports are wasted due to the buffering of instructions in the RUU, and that the number of write ports could be reduced if some of the updates to the register file are discarded. We are not aware of any implementation and experimental results from this work.

In [FS92], the authors make an extensive analysis of the characteristics of register traffic. In their study they conclude that many of the values generated are only used once and that most of the values are used up soon after they have been created, i.e.: between 30-40 instructions later. They also observe that, by using a mechanism to buffer 30 or more instructions, at least 80% of the writes to the register file are unnecessary. The authors briefly discuss compiler mechanisms that could be used to reduce the number of writes to the register files. Our experimental results reported in Section 2.2 and Section 4 are independent from theirs, and our observation that most of the short-lived variables have live ranges within a basic block is new. Our solution strategy and implementation scheme are novel, and can effectively handle programs in the presense of speculative execution.

In [PGH+87], a scheme is proposed where the structure of the reorder buffer is exposed to the compiler in order to increase the performance of the model. The compiler can make use of the reorder buffer to (1) access old register values while new ones are being computed, (2) access the results buffered in the reorder buffer, and (3) select instructions to be undone before they have been committed. Our work has been influenced by their work, but the problem formulation and the solution techniques are different. We have implemented our scheme and reported experimental results.

Finally, the authors are not aware of other works (including the above listed) that explicitly show how to modify the register allocator so as not to assign short-lived variables to physical (architected) registers, thus reducing register requirement and spill cost.

9 Conclusions and Future Work

The rapid advance in VLSI technology has allowed the more recent superscalar designs to use features that not long ago were considered very expensive and difficult to implement on hardware. Aggressive features like the dynamic scheduling of the instructions and the out-of-order issue of instructions supported by the use of reorder buffers are becoming widely used in current and next generations of commercial processors. The introduction of these new features pose new problems for the compiler. In this report we have seen that by allowing the compiler to access the different resources used by the architecture, the implementation of some features in hardware can be simplified and a better utilization of the resources provided can be obtained. In particular, in this report, we have demonstrated how compiler and architecture techniques can be combined to take advantage of the fact that many program variables are short-lived. Our implementation and experimental results have provided ample evidence that such optimization can be effective and may lead to significant performance improvement with relatively little cost.

We plan to continue this study by analyzing how to reduce the number of read ports without affecting the performance of the processor. We also plan to investigate the effect of compile-time variable renaming to increase the efficiency of our analyses and the use of code motion transformations to increase the locality of register accesses.

References

[AA93] Donald Alpert and Dror Avnon. Architecture of the Pentium Microprocessor. *IEEE Micro*, 13(3):11–21, June 1993.

[AAD+93] Tom Asprey, Gregory S. Averill, Eric Delan, Russ Mason, Bill Weiner, and Jeff Yetter. Performance Features of the PA7100 Microprocessor. *IEEE Micro*, 13(3):22–35, June 1993.

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, corrected edition, 1988.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, Houston, Texas, April 1992. Published as Rice COMP TR92-183.

[BW90] H. B. Bakoglu and T. Whitside. RISC System/6000 hardware overview. In Mamata Misra, editor, *IBM RISC System/6000 Technology*, pages 8–15. International Business Machines Corporation, first edition, 1990. Order No. SA23-2619.

[CAC+81] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[CDN92] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 292–300, Portland, Oregon, December 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.

[DA92] Keith Diefendorff and Michael Allen. Organization of the Motorola 88110 superscalar RISC microprocessor. *IEEE Micro*, 12(2):40–63, April 1992.

[DOH94] Keith Diefendorff, Rich Oehler, and Ron Hochsprung. Evolution of the PowerPC Architecture. *IEEE Micro*, 14(2):35–49, April 1994.

[FS92] Manoj Franklin and Gurindar S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 236–245, Portland, Oregon, December 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.

[HDE+92] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[Hsu94] Peter Hsu. Designing the TFP Microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1991.

[Kel75] Robert M. Keller. Look-ahead processors. *Computing Surveys*, 7(4):177–195, December 1975.

[Lai92] Michael Laird. A comparison of three current superscalar designs. *Computer Architecture News*, 20(3):14–21, June 1992.

[Mou93] Cecile Moura. SuperDLX – a generic superscalar simulator. ACAPS Technical Memo 64, School of Computer Science, McGill University, Montreal, Quebec, April 1993.

[PGH+87] A. R. Pleszkun, J. R. Goodman, W.-C. Hsu, R. T. Joersz, G. Bier, P. Woest, and P. Schechter. WISQ: A restartable architecture using queues. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, Pittsburgh, Pennsylvania, June 2–5, 1987. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 15(2), June 1987.

[PS88] A. R. Pleszkun and G. S. Sohi. The performance potential of multiple functional unit processors. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 37–44, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News*, 16(2), May 1988.

[SD94] Peter Song and Marvin Denman. *The PowerPC 604 RISC Microprocessor*. Motorola; IBM Corporation, 1994.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[UJ92] August K. Uht and Darin B. Johnson. Data path issues in a highly concurrent machine. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 115–118, Portland, Oregon, December 1–4, 1992. ACM SIGMICRO and IEEE-CS TC-MICRO.