Conflict-Free Access to Multiple Single-Ported Register Files

Silvia M. Mueller*
University of Saarland
Dept. 14: Computer Science
PF 151150, 66041 Saarbruecken, Germany
smueller@cs.uni-sb.de

Uzi Vishkin†
University of Maryland
Institute for Advanced Studies
College Park, MD 20742-3251, USA
vishkin@umiacs.umd.edu

Abstract

The paper presents a novel static algorithm for mapping values to multiple register files. The algorithm is based on the edge-coloring of a bipartite graph. It allows the migration of values among the register files to keep the number of RAMs as small as possible.

By comparison with the register file design used in the Cydra 5 mini-supercomputer, our approach substantially reduces the number of RAMs. This reduction actually grows with the issue rate. For a system with an issue rate of 6 instructions per cycle, the cost (gate count) of the register files are already cut by half. On a numerical workload, like the Livermore Loops, both designs achieve roughly the same performance.

1. Introduction

Superscalar and very large instruction word (VLIW) architectures achieve a high performance by issuing multiple instructions per cycle, but they require high-bandwidth register files. Current systems have an issue rate of 2 to 6 instructions per cycle, and the rate is still growing (as per [7], for example). That makes the design of fast memories, such as register files, even more important and challenging.

In [10], Rau et al. suggested to store the temporary results in multiple register files, so called *scratch pads*. A viable scratch pad allocation must ensure that *values which are accessed simultaneously reside in distinct scratch-pads.* The compiler has to find an instruction schedule which allows a viable mapping of values to scratch pads. In general, the schedule and the scratch pad allocation have to be adapted iteratively.

*Supported by DFG

†Also affiliated with Tel Aviv University. Partially supported by NSF grant CCR-9416890

The scratch pad design implemented in the processors of the Cydra 5 mini-supercomputer [1, 4] uses a fixed allocation scheme (Section 2). The results of a certain function unit are always mapped to the same scratch pads. Conflict free accesses are ensured by storing the values several times. That results in a simple but very storage intensive register file design.

Our approach (Section 3) aims at providing high register read and write bandwidth with substantially less RAMs. Instead of duplicating the results, our approach allows values to migrate among the scratch pads. Thus, the allocation scheme becomes more complicated again. However, it can be computed efficiently by an edge-coloring algorithm. Section 4 compares the performance and the hardware complexity of the two designs, based on the formal hardware model of [9].

2. The Context Register Matrix CRM

The processors of the Cydra 5 supercomputer [1] implement a VLIW architecture which requires a register file with 12 read and 6 write ports. Instead of a multiport RAM, the processors use a scratch pad design to provide sufficient read/write bandwidth. The allocation scheme of the scratch pads is fixed. That makes it much easier to find a schedule with a viable mapping of the variables.

The core of this scratch pad design is the *context register matrix (CRM)*. The matrix elements are register files. Each write port gets a dedicated row in the CRM. A write access always updates the whole row of the CRM. The register files of a single row therefore have identical contents. Each read port has a dedicated column in the CRM. Let the register files perform two accesses per machine cycle. A CRM with W rows and R columns then permits W conflict-free writes and R reads per cycle. Figure 1 depicts such a CRM.

Since a result is written to all the register files of a dedicated row of the CRM, the instruction word only

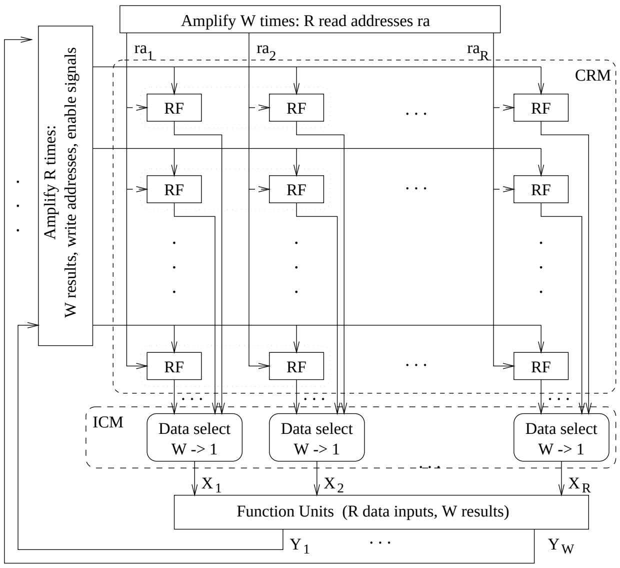

Figure 1. Context register matrix CRM with W rows and R columns. Each matrix element RF is a register file which performs a write and a read per machine cycle. The CRM can serve a set of function units which requires R data inputs and produces W results, in each machine cycle.

specifies a register address per result. Read accesses are more complicated. For each of its R operands X_i, the instruction word specifies a register address ra_i and a CRM row address row_i. The W register files of the i-th column of the CRM are accessed with address ra_i. Based on the row address row_i, one of these W outputs is selected as the output of the read port. This can be done with $(W-1)$ multiplexers arranged in a binary tree. Each read port of the CRM requires such a multiplexer tree. In total, the interconnection circuit ICM of the CRM then comprises the following number of multiplexers:

$$\#\text{ mux in ICM } = R \cdot (W-1) \approx R \cdot W. \qquad (1)$$

The CRM comprises $R \cdot W$ register files. It would be too expensive to implement each register file by a RAM of its own. Since all the register files of a row have identical contents, the engineers of the Cydra 5 decided to use one RAM for two register files each. Thus, two read ports can share a column in the CRM, and the scratch pad design with R read and W write ports can be implemented with

$$\#\text{ RAMs in CRM } = \lceil R/2 \rceil \cdot W \qquad (2)$$

RAMs. These RAMs have to perform three accesses per machine cycle: one write followed by two reads. This optimization cuts the number of RAMs by half, but the CRM scratch pads still scale badly with an increasing number of read and write ports.

3. The Scratch Pad Vector SPV

After describing how to model the scratch pad allocation by graph coloring, we optimize the number of scratch pads and sketch a hardware realization. Our solution builds on the ideas for the implementation of a degenerate family of parallel algorithms in [12].

3.1. General Principles

Given the schedule of an instruction sequence $P = I_1, \ldots, I_p$, our approach proceeds in three steps. It first translates the instruction schedule into a bipartite graph $G(P)$. It then computes an edge-coloring for $G(P)$ and translates the coloring into a viable allocation scheme. An example demonstrating the solution is given in Figure 2.

Instruction Schedule For simplicity, we assume that every instruction I_i comprises several micro-operations which together read up to R operands and write up to W results. The micro-operations of an instruction are executed in parallel: they read the operands, do some processing, and store the results. Each action takes a single cycle. Instructions are executed sequentially.

Access Graph For every instruction I_i of P, the access graph $G(P)$ comprises two nodes R_i and W_i denoting the read and write cycle of I_i. Between any two nodes W_i and R_{i+j}, an edge is added for every variable v,

- if variable v is read by instruction I_{i+j}, and
- if v is written by instruction I_i but not by the instructions $I_{i+1}, \ldots, I_{i+j-1}$.

The access graph $G(P)$ is *bipartite*, because there are no edges between two read or two write nodes.

Allocation Scheme The second step of the algorithm computes an edge-coloring for the access graph, i.e., it assigns colors (labels) to the edges so that any two edges which share an end-point get a different color. Since the access graph is bipartite, an algorithm like the one of [6] can color the edges with labels $1, \ldots, d$, where d denotes the degree of $G(P)$.

Table A

	I_1	I_2	I_3	I_4	I_5
operands	I/O	a, a, b	b, c, c	a, b, e	b, d, f
results	a, b	a, c	e, f	d, I/O	I/O

Access Graph

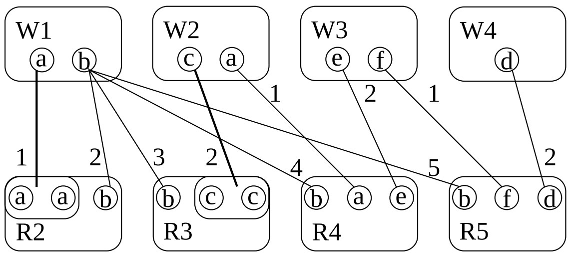

Table B

	write cycle #			
	1	2	3	4
scratch pad 1	a	a	f	
scratch pad 2	b	c	e	d
scratch pad 3	b			
scratch pad 4	b			
scratch pad 5	b			

Figure 2. Access graph $G(P)$ of an instruction sequence $P = I_1, \ldots, I_5$. Table A lists the operands and results of the instructions; "I/O" indicates memory data. Since the graph has degree 5, the edges can be colored with the labels $1, \ldots, 5$. Table B lists the corresponding scratch pad allocation.

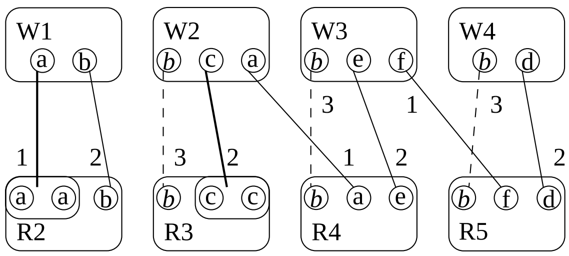

Figure 3. Pruned Access Graph. Values which are used in several cycles are migrated whenever they are read, except for the final read. Edges corresponding to migrated values (in italics) are printed as dashed lines. Pruning reduces the degree from 5 to 3.

Each edge of $G(P)$ corresponds to a write/read access of a value v. The color of the edge specifies the number of the scratch pad which stores v.

3.2. Pruning the Access Graph

Let v be a result of instruction I_i which is read by k instructions. The value v then accounts for k edges adjacent to node W_i, and v must be stored in k scratch pads. Thus, frequently used results blow up the degree of the access graph as well as the number of scratch pads. However, this problem is solved as follows. Except for the final read access, the variable v is read *and* written. Thus, after every read access, the value migrates to a (possibly) different scratch pad.

After this modification, the nodes symbolizing a read still have a degree of at most R. Since every result is only read once before it migrates, the degree of the write nodes is now bounded by $W + B$, where B denotes the maximal number of migrations performed by a single instruction. Thus, the pruned access graph has a degree of at most

$$d \;=\; \max(R, W + B), \tag{3}$$

and d scratch pads are sufficient for the conflict-free execution of the instruction sequence P. In our example (Figures 2 and 3), pruning reduces the degree of the access graph from 5 to 3.

3.3. Hardware Realization of the SPV

Based on this algorithm, we construct a scratch pad design for a set of function units which require R data inputs and produce W results (Figure 4). It is a realistic assumption that the number of operands exceeds the number of results ($R > W$). For the migration of results, we allow B additional write ports. Since we wish that these B ports do not increase the degree of the access graph, we limit their number to

$$B \;\leq\; R - W.$$

This restriction could increase the cycle count, at least for applications with many frequently used results. According to equation (3),

$$\text{\# RAMs in SPV} \;=\; \max(R, W + B) \;=\; R \tag{4}$$

scratch pads (RAMs) are enough to obey the conflict-free rule. Thus, the context register matrix shrinks to a single row (vector) of R register files, which we call *scratch pad vector SPV*. These RAMs perform two accesses per machine cycle: a write followed by a read.

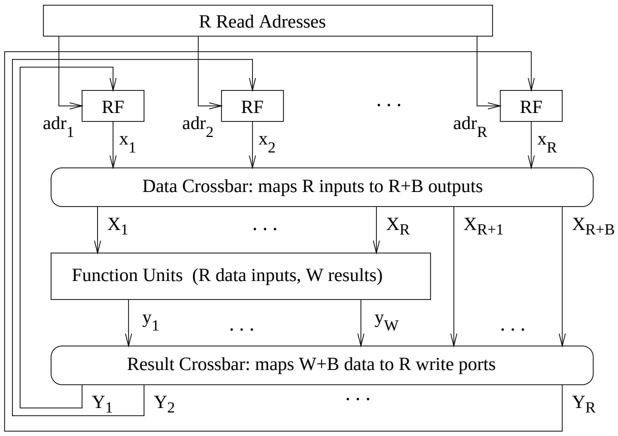

Figure 4. SPV scratch pad design with R register files and B migration paths. The scratch pads serve a set of function units which require R operands and produce W results, in each machine cycle. Each register file RF performs a write and a read access per machine cycle.

Write Access In contrast to the CRM scratch pads, a single write only updates a *single* RAM of the SPV. The edge-coloring mechanism determines which one. As a consequence, the function units no longer have dedicated write ports to the SPV, and therefore, a crossbar is required which distributes the $W + B$ results to the R RAMs of the SPV. For each RAM, this selection can be realized by $(W + B - 1)$ multiplexers arranged as a binary tree. If $R > W+B$, the additional write ports could be used to duplicate some results.

Read Access Even if a result v is used multiple times by the same instruction, v is only read once per instruction. Thus, the scratch values required by a fixed input port of the function units may not necessarily reside in the same RAM of the SPV. A crossbar therefore distributes the R outputs of the register files to the $R + B$ input ports of the function units and migration paths. For each output of the crossbar, this data distribution can be done with $(R - 1)$ multiplexers. The interconnection circuit ICV of the SPV design then comprises the following number of multiplexers:

$$\#\text{ mux in ICV} \;\approx\; R^2 + RW + 2\,RB. \qquad (5)$$

Extensions For methodological reasons, our discussion focused on how to move temporary values among registers to enable conflict-free accesses. However, the data are not limited to values which reside in the register file. So, we must consider ways for loading data from memory in time to avoid delaying instructions which need these data. One possibility is that the compiler could extend our solution for pre-fetching memory data.

4. Quantitative Comparison

This section compares the storage capacity, the hardware complexity and the performance of the two designs. Cost (gate count) and cycle time are evaluated based on the formal hardware model of [9]. Due to lack of space, we must omit most the hardware details, but they can be found on a Web-page[1].

We confine the analysis to reasonable values for the size of the register files and the ratio $f = R/W$ of read and write ports. N denotes the data width of the RAMs and A the number of entries per RAM. The ranges are as follows, where the underlined value denotes the default:

$$\begin{aligned} N &\in \{\underline{32}, 64\} \\ A &\in \{\underline{32}, 64\} \\ f = R/W &\in \{\underline{2}, 3\}. \end{aligned}$$

For current generation superscalar designs [3, 13, 11], f lies between 1 and 3.

The SPV design needs additional write ports for the migration of frequently used results. According to Section 4.3, those results account for at most 30% of the read accesses, at least on the benchmark application. Thus, $B = W/3$ is a reasonable number for the migration paths.

4.1. Storage Capacity

The SPV approach shrinks the context register matrix to a single row. Therefore, one might think that the SPV design can accommodate fewer scratch values, but that is not the case.

In the register matrix approach, a write always updates the whole row of the CRM. Thus, the RAMs of a single row have identical contents. If each RAM has A entries, then the W rows of the CRM hold $A \cdot W$ scratch values.

In our SPV approach, a single write only updates a single RAM. Thus, the R RAMs of the SPV hold $A \cdot R$ scratch values. Since the number of read ports usually exceeds the number of write ports, the SPV scratch pads can actually hold more data than the equivalent CRM scratch pads.

[1] The detailed analysis is available at http://www-wjp.cs.uni-sb.de/~smueller/spv.html

4.2. Hardware Complexity

The two scratch pad designs could be compared purely based on the number of RAMs, i.e., comparing the values of the equations (2) and (4):

$$\begin{array}{rcl} \text{\# RAMs in CRM} & < & \text{\# RAMs in SPV} \\ \iff \quad \lceil R/2 \rceil \cdot W & < & R. \end{array} \tag{6}$$

Under this metric, the CRM approach is only better than the SPV approach for systems which produce $W = 1$ result per cycle. Such a system is neither superscalar nor VLIW.

On the other hand, both scratch pad designs require large interconnection circuits (equations (1) and (5)). The interconnection circuit ICV of the SPV design then uses

$$\frac{\text{\# Mux in ICV}}{\text{\# Mux in ICM}} \approx f + 1 + \frac{2\,B}{W} \tag{7}$$

times as much multiplexers as the interconnection circuit ICM of the CRM design. Thus, the SPV approach reduces the number of RAMs at the expense of a more complicated interconnection circuit. For a given system size (R, W, B), it therefore depends on the cost of a RAM and a multiplexer, whether the SPV design is cheaper than the CRM design.

We now quantify the cost impact within the framework of [9]. Figure 5 depicts the cost (gate count) of the CRM and SPV scratch pad designs as a function of the number of write ports W. The other parameters have their default value, i.e, $N = 32$, $A = 32$, and $f = 2$. In the CRM design, almost all the cost is due to the register files; the interconnection circuit accounts for less than 5% of the total cost.

For the SPV design, the cost of its interconnection circuit ICV grows much faster than the cost of its storage. According to Figure 5, ICV becomes more expensive than the storage when at least $W = 8$ write ports are used.

The diagram further indicates that for more than two write ports ($W > 2$), the SPV design becomes cheaper than the CRM design. This bound is slightly worse than the one derived in equation (6). For $W = 2$, both designs have the same amount of RAMs, but the interconnect of the SPV design is more expensive than the one of the CRM design.

Figure 6 depicts the relative cost C_{CRM}/C_{SPV} of the two scratch pad designs as a function of the number of write ports. The SPV design scales much better as the number of ports increases. For $W = 6$ write ports, switching to the new scratch pad design cuts the cost (gate count) by half. Reducing the size A of the RAMs or increasing the ratio f of read and write ports favors the CRM design; the data width N has virtually no impact.

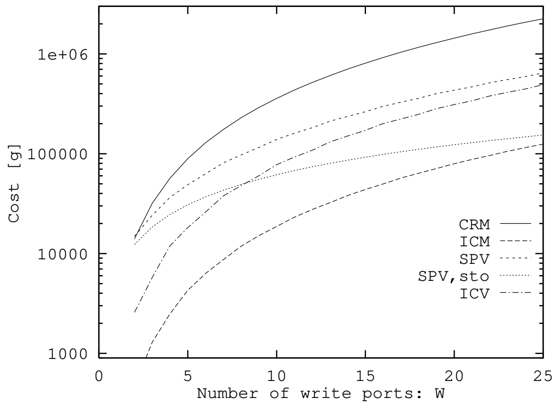

Figure 5. Cost of the storage (sto), the interconnection circuit (ICM, ICV), and of the whole scratch pads for the CRM resp. the SPV design.

4.3. Performance Evaluation

In order to quantify the performance impact of the new scratch pad design, we analyze the run time of the Livermore Fortran Kernels [8] on both designs. Note that the run time of a program on a machine M is the product of a cycle count and the cycle time of M. Changing the scratch pad design can have an impact on both, the cycle count and the cycle time.

Impact on the Cycle Count

The Livermore Fortran Kernels (LFKs) have been compiled for the Cydra 5 with its CRM scratch pads [4], but cycle counts have not been published. For the SPV based design, these data are also missing. Thus, we can only estimate the impact on the cycle count. However, switching to the SPV design can only increase the cycle count due to the following two problems:

1. The compiler cannot anticipate all the memory addresses, and therefore cannot pre-fetch those memory data.

2. Some instructions have to migrate more operands than there are migration paths in the SPV design.

Memory Accesses The LFKs work on vectors and arrays. Except for the kernels 13 and 14 (1D and 2D particle simulation), no indexing is used and the access patterns are very transparent; the vectors are only accessed with union or constant stride. For these type of

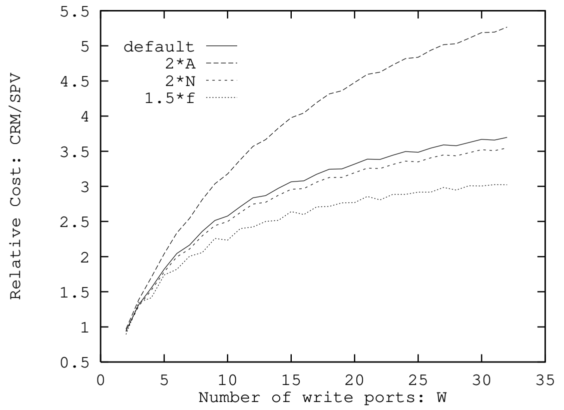

Figure 6. Cost of the CRM design relative to the cost of the SPV design for a several configurations. A denotes the number of RAM entries, N denotes the data width, and f denotes the ration of read and write ports.

accesses, standard vectorizing compilers already provide efficient pre-fetching mechanisms.

In the particle simulations (LFK 13 and 14), most vector accesses use a constant stride, but these two kernels also perform some indexed vector accesses like

```
H[I[j]] := H[I[j]] + 1.0;
```

where the values of the index vector I is not known at compile time. Thus, the indexing makes optimizations of memory accesses, like the pre-fetching of data, almost impossible. Since both design use a static instruction schedule, indexing slows down the SPV based design as much as the CRM based design.

Data Migration Only results used by more than one instruction are migrated. The data flow graphs of the LFKs (e.g., as given in [5]) indicate that, with respect to frequently used results, the kernels fall into one of two classes:

- Each vector element is only referenced once. The pointers to the vectors can be updated whenever they are used for referencing a vector element. Thus, the SPV design can run these kernels without migrating data. The LFKs 1, 3, 5 and 6 fall into this class.

- Some vectors are read several times, but a lot of intermediate results are generated per vector element. Altogether, the frequently used vectors account for less than 30% of all operands. Among others, the LFKs 2, 4, 7, and 8 belong to this class.

In both cases, $W/3$ migration paths should be enough in order to yield the same cycle count as the CRM based design, particularly since the SPV based design can also duplicate some results during write back.

Impact on the Cycle Time

The scratch pad designs can also have an impact on the cycle time of the whole processor. The RAMs of the CRM design perform three accesses per machine cycle, whereas the RAMs of the SPV design only perform two accesses. Let t_{RAM} denote the delay of a RAM access. The two scratch pad designs then require a cycle time of at least

$$t_{CRM} \;=\; 3 \cdot t_{RAM}$$

for the CRM design and

$$t_{SPV} \;=\; 2 \cdot t_{RAM}$$

for the SPV design. Thus, the SPV approach reduces this bound by one third. However, it is unlikely that this also results in a 33% cycle time reduction for the following two reasons. (1) Besides the register files, there are also other time critical components in a processor design. (2) Both scratch pads are designed such that the delay of their interconnection circuit adds to the delay of the function units. Thus, the more complex interconnection circuit of the SPV design can reverse the effect of the faster RAM accesses.

The interconnection circuit ICM of the CRM design only distributes the operands, whereas the SPV design distributes the operands (circuit ICVdat) *and* the results (circuit ICVres). According to Figure 7, the circuits ICM, ICVdat and ICVres are equally fast. Thus, the SPV design roughly doubles the delay of the interconnection circuit. Hiding these additional delays requires the tuning of the SPV design.

The SPV pads are also operated in 3 phases per machine cycle: a write followed by a read followed by an idle cycle which is used for the data distribution. The result distribution is done in the same cycle as the execution of the function units. Since the data distribution (ICVdat) of the SPV design is faster than the RAM access ($t_{RAM} = 22d$), the tuned SPV scratch pads require a cycle time of

$$t'_{SPV} \;=\; 2 \cdot t_{RAM} + t_{ICVdat} \;\leq\; 3 \cdot t_{RAM}$$

which is not slower than the one of the CRM design. After tuning the SPV design, the interconnection circuits of the two scratch pad designs slow down the function units by the same amount. Thus, both designs can be operated at the same clock rate.

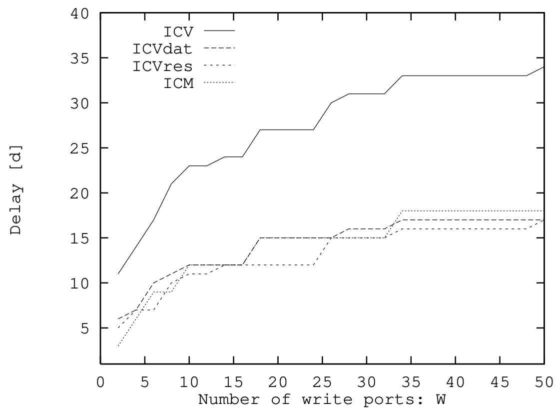

Figure 7. Delay of the two interconnection circuits ICM and ICV as a function of the number of write ports. $ICVdat$ and $ICVres$ denote the delay of ICV for distributing the data resp. results.

Result To sum up the results, the new SPV scratch design can be tuned such that it provides the same read/write bandwidth as the context register matrix CRM but at a much lower hardware cost (gate count).

5. Conclusion

We were able to present a novel static algorithm for a fundamental problem which pertains to both superscalar and VLIW processors. We feel that the idea of moving temporary values among registers is very powerful and should be added to the tool-kit of researchers dealing with designing the software/hardware interface for future processors.

This paper is not the first to present an application of graph coloring allocating temporary values to registers. Following [2], the use of vertex-coloring for modeling such a problem has gotten a lot of attention in the literature (for instance, see the standard text [7]) even though vertex-coloring is NP-complete. While edge-coloring in bipartite graphs can also be used for modeling, it has an important practical advantage: providing efficient algorithms.

Acknowledgment Numerous comments by Shlomo Weiss are gratefully acknowledged.

References

[1] G. Beck, D. Yen, and T. Anderson. The Cydra 5 minisupercomputer: Architecture and implementation. *The Journal of Supercomputing*, 7(1/2):143–180, 1993.

[2] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register alloctaion via coloring. *Computer Languages*, 6:47–57, 1981.

[3] D. Christie. Developing the AMD-K5 architecture. *IEEE Micro*, 16(2):16–26, 1996.

[4] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, 7(1/2):181–228, 1993.

[5] A. Formella. *Performance and Quality of Vectorcomputers for Numerical Applications.* PhD thesis, University of Saarland, Germany, Computer Science Depart., 1992.

[6] H. Gabow and O. Kariv. Algorithm for edge coloring bipartite graphs and multigraphs. *SIAM J. Comput.*, 11:117–129, 1982.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[8] F. McMahon. The Livermore Fortran Kernels test of the numerical performance range. Technical report, Lawrence Livermore National Laboratory, CA, 1988.

[9] S. Müller and W. Paul. *The Complexity of Simple Computer Architectures.* Lecture Notes in Computer Science 995. Springer, 1995.

[10] B. Rau, C. Glaeser, and R. Picard. Efficient code generation for horizontal architectures: Compiler techniques and architectural support. In *Proc. 9th International Symposium on Computer Architecture*, pages 131–139, 1982.

[11] M. Tremblay and J. O'Connor. UltraSPARC I: A four-issue processor supporting multimedia. *IEEE Micro*, 16(2):42–49, 1996.

[12] U. Vishkin and A. Wigderson. Dynamic parallel memories. *Information and Control*, 56:174–182, 1983.

[13] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, 1996.