Partitioned Register File for TTAs

Johan Janssen

Department of Electrical Engineering
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands

Henk Corporaal

Department of Electrical Engineering
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands

Abstract

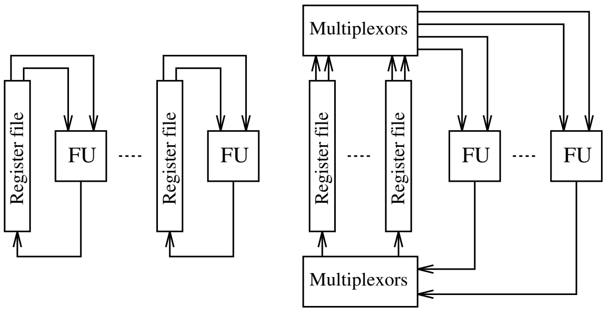
A practical implementation of high performance instruction level parallel architectures is constrained by the difficulty to build a large monolithic multi-ported register file (RF). A solution is to partition the RF into smaller RFs while keeping the total number of registers and ports equal. This paper applies RF partitioning to transport triggered architectures; these architectures are of the VLIW type.

One may expect that partitioning increases the number of executed cycles because it constrains the number of ports per RF. It is shown that these performance losses are small; e.g. partitioning an RF with 24 registers and four read and four write ports into four RFs with 6 registers and one read and one write port gives a performance loss of only 5.8%.

Partitioned RFs consume less area than monolithic RFs with the same number of ports and registers. Experiments show that, if the area saved by partitioning is spent on extra registers, partitioning does, on average, not reduce the performance; it may even result in a small performance gain.

1 Introduction

An architecture that exploits instruction level parallelism (ILP) should be able to read and write multiple register values concurrently; ILP architectures therefore require multi-ported register files (RFs). One approach to exploit ILP is to use a *transport triggered architecture* (TTA), see [1, 2]. Although TTAs reduce the register port requirements with more than 50% as compared to a VLIW with a shared RF [3], the number of required ports can still be substantial for large TTA configurations. However, a large monolithic multi-ported RF increases chip area, causes extra delay and power consumption, and is difficult to realize. This problem can be solved by dividing the monolithic RF into smaller RFs, each with fewer ports, such that the total number of ports remains equal. This results in either a distributed or a partitioned register file configuration [4]; these configurations are shown in figure 1.



a. Distributed register file   b. Partitioned register file

Figure 1: Dividing an RF.

In a distributed RF configuration each function unit (FU), or cluster of FUs, has direct access to one RF only. Access to other RFs requires register copy operations. This configuration has been used in the Multiflow TRACE architecture [5] and within the vector processor CM-5 of Thinking Machines. As shown by Capitanio [6], compilation for a distributed RF configuration is difficult and may result into a large performance loss (up to 75%). Note further, that the register copy operations between RFs may require extra ports on the RFs. These ports are not shown in figure 1a.

A different approach is to partition the RF as shown in figure 1b. Although a partitioned RF provides less connectivity between FUs and registers, when compared to a monolithic RF, each FU has still direct access (for reads and writes) to any register. Therefore, no extra register copies are required. However, still performance losses are to be expected because the limited number of ports per RF may lead to *access conflicts*, i.e. when two or more accesses need the same port.

Partitioned RFs have been used for many years in vector processors. Each vector register in a vector processor can be regarded as a separate RF with only a limited number (often one) read and write ports [4]. Partitioning can also be compared to the use of multiple banks; e.g. a cache split into two banks, each having only one port, can handle two accesses at a time if these accesses are not to the same bank. In both cases the hardware determines and handles the access conflicts. In this paper we propose the use of a partitioned RF, for which access conflicts are handled (i.e. avoided) by the compiler.

Usage of a partitioned RF requires adaptation of the register allocator; it has to distribute the program variables over the partitioned RF. This paper describes several approaches to distribute the program variables over a set of small RFs and measures the performance losses for a set of benchmarks. It will be shown that these performance losses are small. In case we compile for *equal area*, which means that the partitioned RF may occupy the same area as the monolithic RF (with the same number of ports), then partitioning may even result in a small performance gain.

The remainder of this paper is organized as follows. Section 2 describes the architecture and compilation trajectory used within this paper. In section 3 the register allocator without partitioning is described. The methods and their results for allocating registers over multiple RFs, while keeping the number of registers equal, are presented in section 4. In section 5 the results for equal area compilation are given. Finally section 6 concludes this paper.

2 Architecture and compilation trajectory

The tools used for the experiments consists of a hardware framework and a software framework. The main part of the hardware framework is a VLSI(CMOS) processor generator which is build on top of a library of generic VLSI components. With this framework a processor, for an application specific task, can be designed quickly. The processors build, have a transport triggered architecture. The main part of the software framework consists of a front-end compiler and a scheduler.

The register file partitioning is applied to TTAs. A description of TTAs is given in section 2.1. The available ILP is exploited by the software framework. An overview of the compilation trajectory is given in section 2.2. The design space for TTAs is very large. To find a suitable architecture for a given application, a synthesis tool is developed. This synthesis tool is described in section 2.3. Finally, section 2.4 describes the benchmarks used for the experiments and the target architecture; this architecture has been determined by application of the synthesis tool to the benchmarks.

2.1 Transport triggered architectures

TTAs resemble VLIW (very large instruction word) architectures. TTAs and VLIWs contain multiple independent FUs. Rather than programming FU operations directly, TTAs are programmed by specifying the required data transports. As a side effect of these transports, FU operations are triggered. Instead of packing the operations in a single instruction, like VLIWs, TTAs pack multiple transports in a single instruction. An extra level of compiler control is introduced due the programming of these transports. More compiler control enables more code optimization opportunities [3].

A TTA consists of FUs and register files (RFs) connected by a data transport network. An example TTA is given in figure 2. The shown architecture contains one load-store unit, two integer units (ALUs), one floating-point unit (FPU), one immediate unit, one instruction fetch unit, one 8-ported integer RF, a 2-ported floating-point RF and a 2-ported boolean RF. The data transport network has 8 buses; it can handle up to 8 data transports per cycle. This network does not have to be fully connected, although this relaxes the code generation process. It will be clear that the organization of a TTA is such that it is easy to add RFs with arbitrary number of ports.

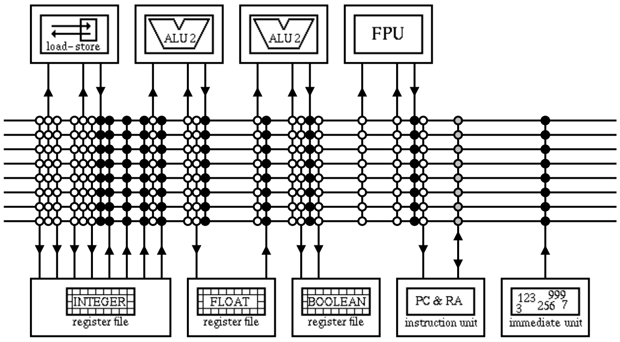


Figure 2: General structure of a TTA.

Each FU contains operand and result registers. Performing an operation with n (source) operands consists of moving n operands to the operand registers of a FU that is capable to perform the operation. The result of the operation can be read from the result register. As an example we show how an add-instruction translates into transports:

$$\texttt{ADD r3,r2,r1} \quad \Rightarrow \quad \texttt{r1} \rightarrow \texttt{O1}_{ADD}\texttt{;}\ \texttt{r2} \rightarrow \texttt{O2}_{ADD}$$
$$\texttt{R}_{ADD} \rightarrow \texttt{r3}$$

First the values of `r1` and `r2` are moved in the operand registers, $\mathrm{O1}_{ADD}$ and $\mathrm{O2}_{ADD}$. After a delay (depending on the latency of the adder) the result can be moved from the result register, R_{ADD}, to `r3`. For more details on TTAs and prototype realizations we refer to [7, 8].

2.2 Compilation trajectory

The code generation trajectory is shown in figure 3. The GNU C/C++ compiler is used as compiler front-end; it is ported to produce sequential TTA code. The scheduler reorders this code and produces parallel TTA code. The scheduler uses, when available, profiling information. This profiling information is obtained from the simulator and used for speculative execution, controlling function inlining, loop unrolling and to compute spilling costs during register allocation. The scheduler is parameterized: all information about the target TTA is read from a machine description file. The scheduler scope is a region, like [9]. More information on the scheduler can be found in [1].

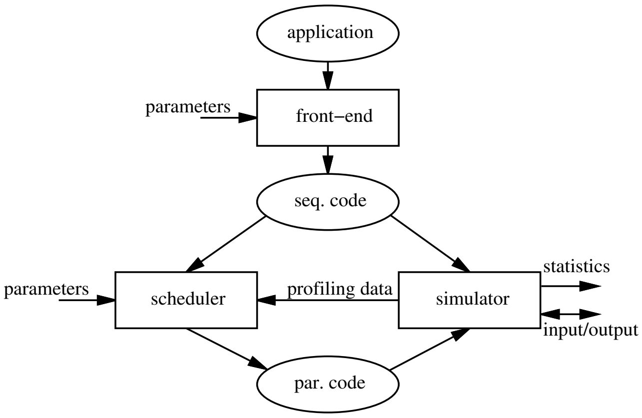


Figure 3: The compilation trajectory.

2.3 Automatic Synthesis

Although we use a TTA template for the design, the design space is still very large. Picking a proper solution (for a specific application) from this design space requires a quantitative analysis of many design points. A synthesis tool is developed that constructs a curve that gives the lowest cost (in area units) for a given performance (as determined by the program execution time) or reverse, gives the best performance for a given cost. Every point on the curve represents an architecture. With the curve the optimal machine architecture for a given cost or performance can be found. A detailed description of this automatic synthesis tool can be found in [1]. Figure 4 gives an example of such a cost-performance curve.

2.4 Benchmarks and target architecture

The goal of our experiments is to measure the performance loss in case RFs are partitioned, and to measure how well the introduced register allocation methods work. We restrict the experiments to mainly integer type of code and research only the partitioning of the integer RF. However, all methods can be applied to floating-point intensive code as well. We use the following 15 benchmarks:

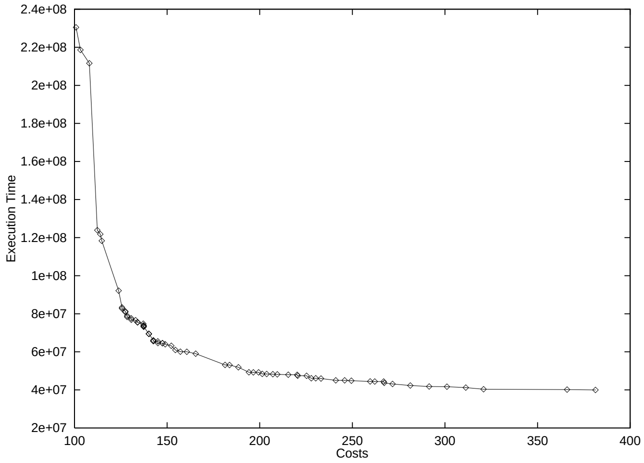


Figure 4: Curve generated by the synthesis tool for benchmark *bison*.

bison, compress, dhrystone, diff, djpeg, expand, flex, mpeg_play, od, sieve, sort, stanford, unexpand, uniq and *wc*.

TTAs have many degrees of freedom; e.g. the number and type of FUs, the data transport capacity, and the number of registers and register ports can be chosen freely. We selected a TTA that has moderate cost and high performance for all benchmarks. To prevent being biased towards a certain TTA configuration, this selection has been made by using the automatic synthesis tool. This tool has been applied to the given set of benchmarks. The resources of the selected architecture are listed in table 1. All FUs, except the floating-point unit, have a single cycle latency and all resources are connected to all buses. The floating-point unit has a latency of three. The number of read ports of the boolean register file is not listed because booleans are read implicitly by *guards*. These guards are used to support conditional data transports. Note, as expected for TTAs that the number of read and write ports of the integer RF is less than would be required for a VLIW with the same number of FUs. If we would increase the number of ports the benchmarks would barely show any performance loss when partitioning the RF (the architecture would be 'over-designed' such that access conflicts hardly occur).

3 Register allocation for a monolithic register file

Register allocation is the problem of finding the most efficient mapping, in terms of execution time, of program variables (also called *symbolic registers*) to the registers of

Table 1: The target TTA used for the experiments.

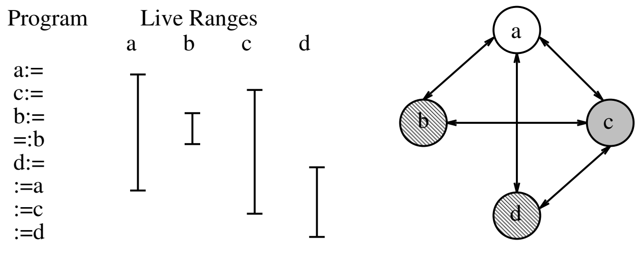
Resource	**Quantity**
# Buses	8
FUs	
• # Integer FUs	3
• # Floating-point FUs	1
• # Load-Store FUs	2
Integer register file	
• # register	24
• # write ports	4
• # read ports	4
Floating-point register file	
• # registers	8
• # write ports	1
• # read ports	1
Boolean register file	
• # registers	8
• # write ports	2
Immediate units	2

the target machine (i.e. the *machine registers*). Register allocation is a complex task and has a severe impact on the resulting code. A commonly used method is graph coloring. Before explaining graph coloring some definitions are introduced:

- A variable is defined at a point in a program when a value is assigned to it.
- A variable is used at a point in a program when its value is referenced in an expression.
- A variable is said to be live at a point if it has been defined earlier and will be used later.
- The *live range* of a variable is the execution range between definitions and uses of a variable. Figure 5a shows a program-segment and its associated live ranges.

The number of variables exceeds normally the number of machine registers. Fortunately not all live ranges of the variables overlap. This means that we can map multiple variables to a single machine register as long as the corresponding live ranges do not overlap. A proper register allocation is the mapping of the variables of a program onto the available machine registers such that no machine register is assigned to any two variables that are live simultaneously.

To find an efficient mapping an *interference graph* is constructed. The interference graph of a program consists of a vertex for each variable and edges between any two



a) Program with live ranges b) Interference graph

Figure 5: Graph coloring example.

vertices if and only if the two variables associated with the vertices are live simultaneously. Figure 5b shows the interference graph of the program-segment of figure 5a. Variables b and d can be allocated in the same register but b and a cannot. A proper register allocation for the program-segment of figure 5 requires three colors. The problem of register allocation can be described as the problem of finding a proper vertex coloring for the interference graph with k colors, where k is the number of available machine registers.

Of course it is not always possible to map all variables to machine registers. This occurs when there are more variables simultaneously live then there are machine registers. In this case the interference graph cannot be colored and a variable must be spilled to memory. Whenever a use of the spilled variable occurs, the variable is reloaded.

The first implementation of a graph coloring allocator was described by Chaitin et al. [10]. In [11] an improvement of this method is described. Our register allocator is based on that method. Figure 6 shows the basic scheme of our allocator. It contains six phases:

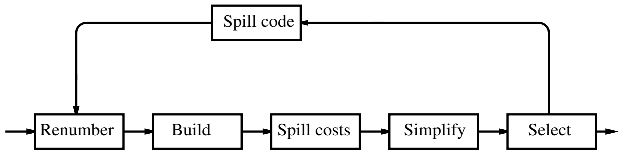


Figure 6: The structure of the register allocator.

Renumber: The first phase finds all live ranges in a procedure and numbers (renames) them uniquely.

Build: This phase constructs the interference graph.

Spill Costs: In preparation for coloring, a spill cost estimate is computed for every live range. The cost is simply the sum of the execution frequencies of the transports that define or use the variable of the live range.

Simplify: This phase removes nodes with degree $< k$ in an arbitrary order from the graph and pushes them on a stack. Whenever it discovers that all remaining nodes have degree $\geq k$, it chooses a spill candidate. This node is also removed from the graph and optimistically pushed on the stack, hoping a color will be available in spite of its high degree.

Select: Colors are selected for nodes. In turn, each node is popped from the stack, reinserted in the interference graph and given a color distinct from its neighbors. Whenever it discovers that it has no color available for some node, it leaves the node uncolored and continues with the next node.

Spill Code: If there remain uncolored nodes then the phase *Spill Code* is invoked. This phase inserts spill code for the live ranges of all uncolored nodes. After spilling the register allocator starts again with the first phase in an attempt to color the modified code-segment with k colors.

Some symbolic registers must be mapped on a specific machine register; e.g. in our compiler the stack-pointer is mapped on r1. These registers get their color in the simplify stage instead of being pushed on the stack. The other machine registers are equally divided in caller-saved and callee-saved registers. The allocator computes the caller-saved and callee-saved cost. The caller-saved cost for the symbolic registers is computed when they have live-ranges across procedure calls. The cost per symbolic register is twice the execution frequency of this call. The callee-saved cost of a symbolic register is twice the execution frequency of the procedure to which the transport of the symbolic register belongs. With these two costs in mind the allocator chooses a machine register.

Scheduling can be done either before or after register allocation. When scheduling of the transports is carried out after register allocation the selection of machine registers may limit the possibilities to reorder transports due to false dependences introduced with the reuse of machine registers. On the other hand, when scheduling transports precedes register allocation, the span of live values may increase: more symbolic registers are simultaneously live and hence more spill code will be inserted. Another disadvantage of this method is that the inserted spill code requires an extra scheduling step.

Our back-end performs register allocation before scheduling. To avoid a large performance loss due to false dependences we changed the select phase such that it tries to prevent the introduction of false dependences; this is based on [12]. A cost function for the mapping of a symbolic register to a machine register is given. This cost function is a measure for the introduction of false dependences. The heuristic for choosing a machine register for a symbolic register becomes:

$$\begin{aligned} h1 = \quad & min(fdep(col_1, reg) + caller(reg), \\ & fdep(col_2, reg) + callee(reg)) \end{aligned} \tag{1}$$

where col_1 represents a caller saved machine register, col_2 represents a callee saved machine register and reg represents the symbolic register. The function $fdep(col_x, reg)$ returns a value that is a measure for the introduction of false dependences when mapping col_x to reg. The functions $caller(reg)$ and $callee(reg)$ return the cost for respectively mapping reg on a caller saved register or on a callee saved register. During the select phase the first available color (register) that achieves the minimum of $h1$ is taken.

4 Register allocation for partitioned register files

Partitioning the RF does not induce much changes to the scheduling process. The existing TTA scheduler knows exactly which ports, connections, and transport buses are available between any source and destination. This means that only the register allocator, as described in the previous section, has to be adapted such, that it distributes the registers over the partitioned RFs. This distribution should result in a minimal performance loss. In this section four distribution methods are described and evaluated. The first two methods use a simple distribution, while the latter two are based on more advanced heuristics.

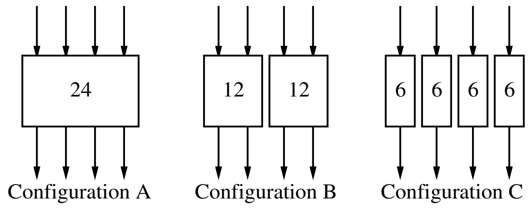


Figure 7: The register file configurations used for the experiments.

Together with the distribution methods, the measurement results of applying these methods are given. All measurements assume a constant number of integer registers (24) and a constant number of ports on the integer RF (4 read and 4 write ports). This is depicted in figure 7, which shows three configurations; A, B and C, of the integer RF as used for the measurements. The results of the measurements for configurations B and C are listed relative to the results of configuration A; in formula:

$$cycles_{relative} = \frac{cycles\ with\ partitioning}{cycles\ without\ partitioning} \cdot 100\%$$

The remainder of this section is organized as follows. In section 4.1 simple distribution methods and their results are presented. In the next section more advanced methods are described and their results are given. Finally, in section 4.3 the discussed methods are evaluated.

4.1 Simple distribution methods

Let us assume that all registers fit into a single address space; e.g. if there are four RFs with 6 registers each then we assume that we can address each register with an address between 0 and 23. The problem now is how to distribute the register addresses over the RFs. The simplest solution is to assign the register addresses 0..5 to RF0, 6..11 to RF1 and so on. This distribution is called the **vertical distribution**, see figure 8. The results of this distribution are listed in table 2.

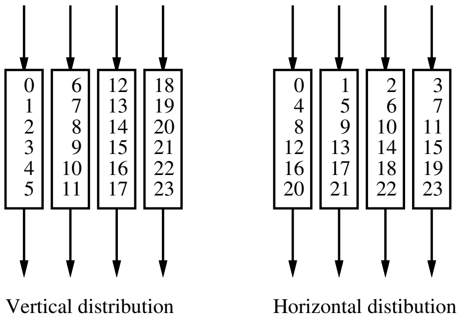


Figure 8: Distribution of the register numbers over the register files (for configuration C).

The second column of this table lists for each benchmark the number of operations/cycle for configuration A; this column gives an indication of the quality of the compiler used. The third column, $v(B)$, lists the results for a vertical distribution using configuration B and column four, $v(C)$, list the results for this distribution using configuration C. The performance loss for a partitioning in four parts is quite large. This is because the register allocator picks the first available register and therefore registers with a low number are picked relatively often. These registers are however located in the first RF and must share the same number of ports. This results in many access conflicts during the scheduling process.

The **horizontal distribution**, see figure 8, avoids this problem because it distributes the registers so that the number of accesses to each RF are more uniformly distributed. The results of this distribution are listed in table 2 as well. Column $h(B)$ lists the results for a horizontal distribution using configuration B and column $h(C)$ list the results for this distribution using configuration C. From the results we see that this distribution performs much better; the average performance loss with a partitioning into four parts is only 7.2%.

Table 2: Results of the vertical and horizontal distributions.

benchm.	**opc**	$v(B)$	$v(C)$	$h(B)$	$h(C)$
bison	2.7	102.1%	113.2%	101.6%	105.2%
compress	2.2	103.9%	106.2%	103.6%	105.3%
diff	2.1	101.6%	124.7%	100.7%	101.8%
dhrystone	2.9	101.9%	119.0%	100.4%	106.7%
djpeg	2.7	104.7%	116.0%	102.7%	105.2%
expand	3.1	104.1%	122.7%	104.1%	115.5%
flex	2.7	103.0%	112.3%	100.9%	104.3%
mpeg_p.	2.7	104.9%	115.0%	102.2%	106.6%
od	3.1	104.0%	117.1%	102.4%	107.9%
sieve	2.7	107.7%	119.0%	100.0%	112.2%
sort	3.1	101.8%	123.0%	102.9%	107.2%
stanford	2.9	103.0%	117.6%	100.6%	106.4%
unexpand	2.9	100.3%	114.6%	103.3%	107.3%
uniq	3.3	101.2%	117.9%	102.6%	110.4%
wc	2.9	100.0%	100.5%	100.0%	106.2%
average	2.8	102.9%	115.9%	101.9%	107.2%

4.2 Advanced distribution methods

The two distribution methods described in this section try to improve the results of the horizontal distribution. To accomplish this the heuristic as described in equation 1 is modified.

Data independence heuristic: The data independence heuristic checks whether two transports are independent. When transports are independent, the scheduler may schedule them in the same cycle. However the scheduler cannot schedule them in the same cycle if there are more transports from or to the same RF than there are ports on that RF. The scheduler has to delay one or more transports which may lower the performance.

The data independence heuristic tries to avoid this by mapping the variables whose transports may be scheduled in the same cycle, in different RFs. A square matrix is used with sizes equal to the number of symbolic registers. By inspecting the dependence graph we select all pairs $src1 \rightarrow dst1, src2 \rightarrow dst2$ of independent transports. For the symbolic registers of every pair of independent transports, data independent and resource independent, the entries $(src1, src2)$ and/or $(dst1, dst2)$ of the matrix are updated with:

$$\frac{p(BB_1, BB_2) * freq_1 * freq_2}{dist(transport_1, transport_2)}$$

The function p returns the probability that basic block BB_1

Table 3: Results of the heuristics.

benchmark	$h2(B)$	$h2(C)$	$h3(B)$	$h3(C)$
bison	102.2%	102.0%	100.4%	103.4%
compress	100.1%	110.1%	101.6%	103.7%
diff	99.9%	101.0%	99.9%	100.5%
dhrystone	98.9%	111.9%	99.3%	104.5%
djpeg	101.7%	104.0%	101.7%	107.3%
expand	104.5%	100.0%	104.5%	120.2%
flex	101.2%	103.5%	101.4%	104.0%
mpeg_play	101.3%	112.6%	99.7%	103.4%
od	103.5%	109.3%	104.3%	107.1%
sieve	101.4%	106.2%	101.7%	105.3%
sort	100.9%	107.5%	100.2%	105.3%
stanford	101.7%	105.1%	100.5%	107.6%
unexpand	103.1%	107.0%	96.7%	103.7%
uniq	102.9%	108.1%	102.1%	106.4%
wc	105.5%	109.5%	105.8%	103.9%
average	101.9%	106.5%	101.3%	105.8%

to which $transport_1$ ($src1 \rightarrow dst1$) belongs is preceded or succeeded by basic block BB_2 to which $transport_2$ ($src2 \rightarrow dst2$) belongs (BB_1 and BB_2 may, of course, be equal). The values $freq_1$ and $freq_2$ are the frequencies of these basic blocks. The probabilities and the frequencies are obtained from the profiling information. The function $dist(transport_1, transport_2)$ returns the distance between the two transports, the longer the distance the smaller the chance that the two transports are scheduled in the same cycle and the less important for the heuristic. The higher the frequency, the more important the transports are for the total schedule.

The select phase selects the best machine register for a symbolic register in the same way as for the previous heuristic, h1. The new heuristic is:

$$h2 = \\ min(fdep(col_1, reg) + caller(reg) + Indep(col_1, reg), \\ fdep(col_2, reg) + callee(reg) + Indep(col_2, reg))$$

where $Indep(col_x, reg)$ returns a value that is a measure for the number of independences when reg is placed in the RF of col_x.

The results of applying heuristic h2 are shown in table 3; the column $h2(B)$ lists the results for configuration B, and column $h2(C)$ for configuration C. Configurations B and C show average performance losses of 1.9% and 6.5% respectively.

Intra-operation heuristic: The intra-operation heuristic tries to avoid multiple reads within a single operation from a single port-limited RF. We will illustrate this problem with an example: suppose an RF has only one read port. An operation usually needs two operands. If both operands are located in the same RF then it is impossible for the scheduler to schedule the two transports in the same cycle. If however the transports are located in different RFs then they can be scheduled in the same cycle and hence gain performance.

Again a square matrix is used with sizes equal to the number of symbolic registers. When two read transports occur in the same operation, called a *clash*, the entry $(operand1, operand2)$ is updated with the frequency of the basic block in which this operation occurs. The reason to use the frequency is that relatively often executed code-segments must be scheduled more efficiently to obtain good overall performance. Avoiding a clash between two read transports from the same RF can result in a more efficient schedule.

The new heuristic in the select phase becomes:

$$h3 = \\ min(fdep(col_1, reg) + caller(reg) + Intra(col_1, reg), \\ fdep(col_2, reg) + callee(reg) + Intra(col_2, reg))$$

where $Intra(col_x, reg)$ returns a value that is a measure for the number of clashes when reg is placed in the RF of col_x. The results of heuristic $h3$ are listed in the fourth and fifth column of table 3. As indicated, this heuristic performs better, on average, than heuristic h2.

4.3 Evaluation

The experiments with the various heuristics show that a small loss in performance, due to access conflicts, has to be accepted when partitioning the RF. The distribution method has a large impact on the performance; e.g. the vertical distribution results in a performance loss of more than 15.9% for configuration C while the horizontal distribution results in a performance loss of only 7.2%.

Usage of heuristics may reduce the performance loss of the horizontal distribution somewhat, especially for configuration C. Heuristic h3 turns out to be better than heuristic h2. This is caused by the fact that there are many transports that can be scheduled in the same cycle but only few actually do. Much information collected by heuristic h2 is therefore not useful and can make things even worse, as shown by individual benchmark results. Some of the results show an improvement. For example when using heuristic $h2$ for configuration B the benchmarks *diff* and *dhrystone* show a small performance gain. The cause of this gain is that the heuristics of the scheduler have a somewhat unpredictable behavior. This behavior can result in a slightly better result. A combination of heuristics h2 and h3 has also been researched; however, it did not result in a further improvement.

The performance results shown in this section are much better than the results reported for distributed RFs in [6]; the reported performance losses for distributed RFs ranged from 4% to 40% for two partitions and from 15% to 75% for four partitions. The reason is twofold. First, partitioned RFs do not require extra copy register operations between registers of different partitions. Second, in TTAs many results of operations, especially those which appear to be critical, are directly forwarded from FU result to FU operand register without being written to or read from the RF; they therefore do not cause RF access conflicts.

The access times of the RFs can, according to [6], be modeled as at least a linear function of the number of output ports. Furthermore, the access times are linear with the number of registers, see [13]. According to these results, smaller RFs with fewer ports are faster. If the register access time constrains the achievable cycle time of a processor and therefore critical in the overall performance of the system then the performance loss is even smaller than listed in the tables 2 and 3.

5 Compiling for equal area

A partitioned RF occupies less chip-area for the same number of registers. To compensate for that we add more registers to a partitioned RF and redo the comparison between a monolithic and a partitioned RF. In section 5.1 a model is presented for the chip-area of a multi-ported RF. This model is used to equalize the area of a monolithic and a partitioned RF. In section 5.2 the performance of these RFs is compared.

5.1 Area model

To make a comparison in area between the various RF configurations we use a simple RF implementation model; it is shown in figure 9. This figure shows a 1-bit, 4-transistor SRAM cell with two ports. An RF with 24 registers of 32-bit will contain 32 of those cells in a row (forming one word or register) and 24 registers in a column.

Each port requires two transistors per SRAM cell. To address a register, address decoders are used. Every port needs its own address decoder. For this decoder we assume that a dynamic row decoder circuit is used. The number of transistors for this circuit per row per port is: the number of address bits plus an additional 9 transistors [14]. In our model we assume an additional number of 17 transistors per bitline for the bitline drivers(2) and sense amplifiers(15) [13]. The total transistor count is then given by:

$$
\begin{aligned}
\#trans \;=\; & \overbrace{4 * \#reg * reg.width}^{SRAM\ cells} + \\
& \overbrace{2 * \#ports * reg.width * \#reg}^{register\ ports} + \\
& \overbrace{(\#address\ bits + 9) * \#ports * \#reg}^{row\ decoders} + \\
& \overbrace{\#ports * reg.width * 17}^{bitline\ drivers + sense-ampl.} \qquad (2)
\end{aligned}
$$

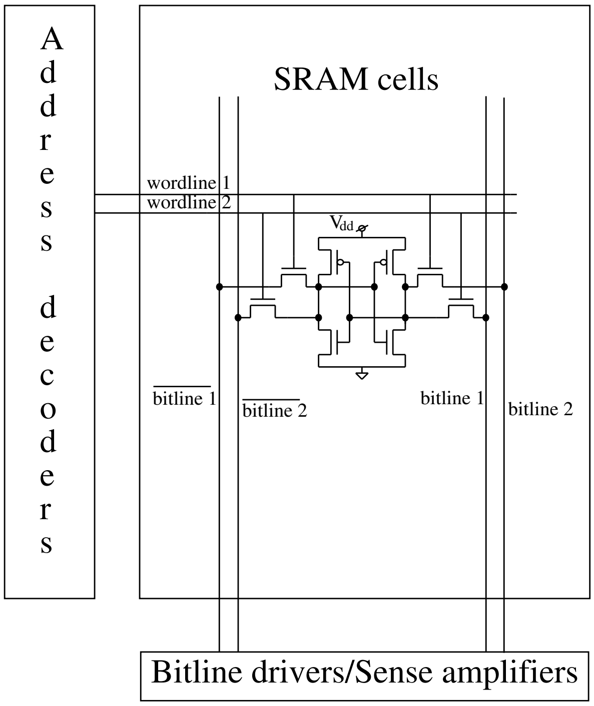


Figure 9: Implementation of a register file (only one bit has been drawn).

where $\#trans$ is the number of transistors, $\#reg$ is the number of registers, $reg.width$ is the width of a register in bits, $\#ports$ the sum of the number of read and write ports and $\#address\ bits$ the number of bits needed to address a register; $\#address\ bits\ = \lceil \log_2\ \#reg \rceil$.

Based on equation 2, and assuming that the number of transistors is an approximate measure for the circuit area, the area of a multi-ported RF can be determined. Figure 10 draws the area for three RF configurations, a monolithic RF, one with two partitions, and one with four partitions in units of an 8-ported monolithic RF with 24 registers (i.e. relative to configuration A). Each configuration has a total of eight ports. From this figure we see that configuration A has an area 1.5 times the area of configuration B and 2.0 times the area of configuration C.

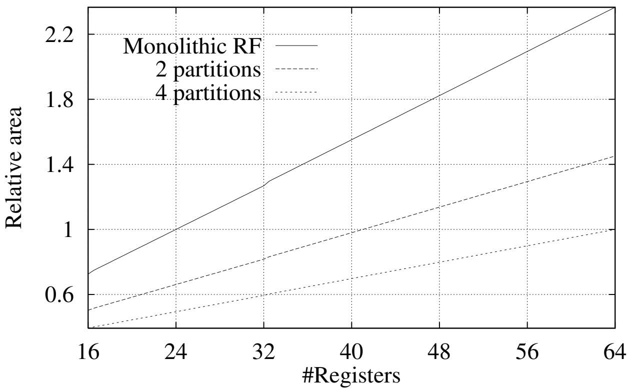


Figure 10: Area of various 8-ported RF configurations relative to the area of a monolithic 8-ported RF with 24 registers.

5.2 Compiling for equal area

In this section we will spend the area saved by partitioning on extra registers, and measure again the relative performance of the benchmarks for the various configurations. Figure 10 shows that, in order to equalize the area of the three configurations, configuration B may use two partitions of 20 registers each, and configuration C can have four partitions of 16 registers each; thus, configuration A still has 24 registers, configuration B gets a total of 40 registers and configuration C gets 64 registers. The performance measurements of these enlarged configurations, called B' and C', will use the heuristic $h3$; with this heuristic the best results were obtained. The results are listed in table 4.

From the table we conclude that with equal area compilation the performance losses caused by partitioning are negligible. Some benchmarks show disappointing results, though; this is because these benchmarks probably have enough on 24 registers and adding more registers will not help to gain performance. For these benchmarks it is more profitable to spend the saved area on other resources like function units, more cache, etc.

Table 4: Results of equal area scheduling.

benchmark	$h3(B')$	$h3(C')$
bison	101.7%	100.4%
compress	101.5%	105.3%
dhrystone	96.6%	103.6%
diff	99.5%	99.1%
djpeg	98.1%	101.6%
expand	104.5%	114.6%
flex	98.3%	99.0%
mpeg_play	76.5%	75.2%
od	98.3%	100.0%
sieve	97.5%	106.2%
sort	95.0%	97.7%
stanford	97.5%	100.0%
unexpand	103.2%	106.7%
uniq	95.9%	88.6%
wc	106.2%	105.8%
average	98.0%	100.3%

6 Conclusions

Partitioning of register files (RFs) is an attractive solution to overcome one of the design problems, namely the difficult implementation of monolithic RFs with many ports, of processors which exploit large amounts of instruction level parallelism. Partitioning may however increase the number of executed cycles due to RF access conflicts. In this paper we have shown that RF partitioning, when applied to TTAs, does not have to result in a large performance loss; e.g. using a horizontal distribution of the registers among the partitions results in an average performance loss of 1.9% for two and 7.2% for four partitions. Using heuristics this loss may even be reduced to 1.3% and 5.8% respectively.

These results are much better than the ones reported for distributed RFs. This is caused by the fact that partitioned RFs do not require the insertion of extra register copy operations. Furthermore, TTAs add an additional advantage: many critical operations do not read to or write from the RF; they forward their results directly from FU output to FU input.

A partitioned RF costs less area. If we compensate for this, by adding extra registers to the partitioned RF, the performance losses caused by partitioning have been shown to disappear; the usage of two partitions may even result in a small performance gain.

Finally, we summarize the advantages of using a partitioned RF versus a monolithic RF: 1) a partitioned RF is easier to realize due to the low complexity of the components; 2) the approach is scalable: the RFs can be duplicated to fit the architecture requirements; 3) one can design with available low cost SRAM cells; 4) the required chip area may reduce; 5) the access time of an RF with fewer ports is smaller and 6) the performance loss is very small. Since the trend is towards the exploitation of ILP, it is likely that the RF becomes a larger bottleneck. Our approach will offer good performance as register (bandwidth) requirements double and redouble from current needs.

References

[1] Henk Corporaal and Jan Hoogerbrugge. Code generation for Transport Triggered Architectures. In *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[2] Jan Hoogerbrugge and Henk Corporaal. Transport-triggering vs. operation-triggering. In *Lecture Notes in Computer Science 786, Compiler Construction*, pages 435–449. Springer-Verlag, 1994.

[3] Jan Hoogerbrugge and Henk Corporaal. Register file port requirements of transport triggered architectures. In *MICRO-27*, Santa Clara, December 1994.

[4] Corinna Grace Lee. *Code Optimizers and Register Organizations for Vector Architectures*. PhD thesis, Univ. of California, Berkeley, May 1992.

[5] Michael Schuette and John Shen. Instruction-level experimental evaluation of the multiflow trace 14/300 vliw computer. *The Journal of Supercomputing*, 7(1/2):249, May 1993.

[6] Andrea Capitanio, Nikil Dutt, and Alexandru Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Tradeoffs. In *MICRO-25*, pages 292–300, Portland, December 1992.

[7] Henk Corporaal and Hans (J.M.) Mulder. MOVE: A framework for high-performance processor design. In *Supercomputing-91*, pages 692–701, Albuquerque, November 1991.

[8] Henk Corporaal and Paul van der Arend. MOVE32INT, a Sea of Gates realization of a high performance Transport Triggered Architecture. *Microprocessing and Microprogramming*, 38:53–60, September 1993.

[9] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proc. of the ACM SIGPLAN 1991 conference on Programming Language Design and Implementation*, pages 241–255, June 1991.

[10] Gregory J. Chaitin et al. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[11] Preston Briggs. *Register Allocation via Graph Coloring*. PhD thesis, Rice University, April 1992.

[12] Shlomit S. Pinter. Register Allocation with Instruction Scheduling: a New Approach. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 248–257, June 1993.

[13] Tomohisa Wada, Suresh Rajan, and Steven A. Przybylski. An analytical access time model for on-chip cache memories. *IEEE journal of solid-sate circuits*, 27:1147–1156, August 1992.

[14] A. J. van de Goor. *Testing Semiconductor Memories: Theory and Practice*. John Wiley & Sons, 1991.