# Register Allocation for Processors with Dynamically Reconfigurable Register Banks

Ralf Dreesen, Michael Hußmann, Michael Thies and Uwe Kastens
Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn, Germany
{rdreesen,michaelh,mthies,uwe}@uni-paderborn.de

March 11, 2007

## Abstract

This paper presents a method for optimizing register allocation for processors which can reconfigure their access to different register banks to increase the number of accessible registers.

Register allocation like the graph coloring method of Chaitin aim at keeping a large number of values in registers and minimize the amount of necessary spill code.

We extend that method in two directions: First, reconfiguration code is needed to switch between registers banks. Those code costs, too, are minimized. Second, our method chooses the values to live together in a register bank, such that reconfiguration costs are reduced.

For the latter task, a novel program analysis is proposed, which applies a DFA to yield information on register access patterns.

The register allocation method can be extended for a multi-core processor with a shared reconfigurable register bank.

## 1 Introduction

This paper proposes a method, which performs a *register allocation* for a reconfigurable register bank. The proposed method inserts *reconfiguration instructions* into the program code, to make different sets of registers accessible. It is also suitable for processors with multiple synchronous cores, as will be shown at the end of this paper.

To discuss the register allocation method, the properties of the reconfigurable register bank are described first.

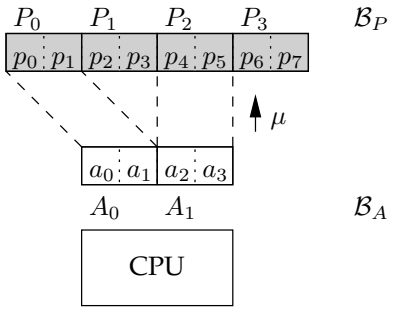


Figure 1: Schematic reconfigurable register bank

**Reconfigurable Register Bank** The reconfigurable register bank consists of a large set of registers. The registers can not be accessed directly, but only through a smaller set of register frames, which are actually encoded as operands in instructions. In the following, the registers which actually store values are called *physical registers* and register frames are called *architectural registers*.

**Reconfiguration** The mapping of physical registers into architectural registers is changed explicitly at runtime by *reconfiguration instructions*, which are part of the executed program.

To reduce the number of reconfiguration instructions, each change of a mapping affects a block of registers, instead of just one register. Therefore, the physical and architectural registers are divided into equal sized blocks of registers. Hence, a reconfiguration instruction maps a *physical register block* into an *architectural register block*.

**Terminology** The schema of a reconfigurable register bank is shown in figure 1. Physical registers are

denoted by $p_i$ and architectural registers by $a_i$. The symbols $P_i$ refer to physical blocks and $A_i$ to architectural blocks. The set of all physical blocks is given by $\mathcal{B}_P$ and the set of architectural blocks by $\mathcal{B}_A$.

The mapping of physical blocks into architectural blocks is given by the function $\mu$, which maps an architectural block to a physical block. In figure 1 $\mu$ maps $A_0$ to $P_0$ and $A_1$ to $P_2$. If the mapping of an architectural block can not be predicted at compile time, it is said to be mapped to bottom ($\perp$).

**Overview** The proposed register allocation method extends the graph coloring approach of Chaitin [1]. Therefore, affinities between virtual registers $v_i$ and physical blocks $P_j$ are computed (section 3.1). Affinities estimate the number of reconfiguration instructions that are avoided by assigning $v_i$ to a physical register of $P_j$. The affinities are used during coloring as a secondary criterion for choosing a physical register.

Reconfiguration instructions are inserted into the program using the "as late as possible" strategy, i.e. right before an access to an unmapped physical register (section 3.2). An architectural block for the mapping is chosen according to the replacement strategy "Latest Used Again". In order to reuse mappings that are established in preceding basic blocks, this paper presents a method to define mapping conventions between basic blocks.

We discuss our register allocation method for a single processor first. Section 3.3 describes minor modifications needed for multi-core processors.

# 2 Related Work

Our register allocation method builds on classic graph coloring as introduced by Chaitin [1]. Chaitin reduces the problem of assigning virtual registers to physical registers to a graph coloring problem. We exploit some freedom that is left in this algorithm, to model reconfiguration of register banks and to minimize reconfiguration code. Reconfiguration follows the replacement strategy proposed by Belady [2]. Originally a paging strategy for MMUs, it is optimal with respect to the number of page replacements. It can also be applied as a register allocation strategy for straight-line code. Driven by limited information on future register accesses, we use Belady's strategy to choose the best physical block to be replaced. This reduces the number of reconfiguration instructions.

In the following, we describe two restricted forms of register reconfiguration, which are used in widespread conventional processors. After that, we outline two scientific reconfigurable processor designs and relate their associated register allocation methodologies to our approach.

**Register Renaming** Register renaming [3] is a hardware mechanism to avoid register access constraints caused by pending memory accesses. If the content of a register $r$ is to be stored, it must not be overwritten by a succeeding instruction, until the store is completed. Register renaming circumvents this restriction by binding a different physical register to $r$. This way, the pending store instruction can access the old value of $r$, while the new value is stored in another physical register.

In contrast to our approach, register renaming does not increase the number of compiler-exploitable registers, but is limited to shortening register life times by a fixed amount.

**Register Windowing** Register windowing, as described in [4], is a mechanism to make different sets of registers accessible using a sliding window. Its primary purpose is to avoid memory accesses caused by register save and restore code on function calls. Register windowing constitutes a special case of our model: Only adjacent physical blocks can be mapped into architectural blocks. Moreover, repeated `save` and `restore` instructions might be necessary, to make a physical block accessible. Thereby, it is not possible to select a *specific* physical block for replacement. As a result of these restrictions, register windowing is not viable for code with interspersed accesses to many registers.

**Register Connection** Kiyohara et al [5] propose a family of register architectures with two classes of registers, namely the *core registers* and the *extended registers*. With their most elaborate architecture, core registers can be accessed without additional effort, whereas every access to an extended register must be preceded by a reconfiguration instruction. The register allocation method assigns heavily accessed virtual registers to core registers, whereas the remaining registers are assigned to extended registers. In contrast to our register architecture, registers are not organized in blocks and thus an affinity analysis is not necessary. If the number of core registers is exceeded, many reconfiguration instructions are likely to be inserted to prepare access to extended registers.

**Windowed Register File** The register architecture as proposed by Ravindran et al [6] divides physical registers into banks, where only one bank at a time can be accessed. As a result, additional copy instructions (inter-window-moves) are necessary to allow simultaneous access to registers in different banks.

To estimate reconfiguration and copy costs, affinities are used to partition the registers into banks.

The affinities estimate the reconfiguration costs only roughly. Using a special version of our analysis, the reconfiguration costs could even be determined exactly.

The architecture of Ravindran et al lends itself better to an exact estimation of reconfiguration costs, but incurs the aforementioned inter-window-move instructions.

# 3 Register Allocation

In the register allocation phase of the compiler, the virtual registers as used in the abstract machine program are assigned to the physical registers of the processor. With a reconfigurable register bank, the same task is to be solved. Hence, the virtual registers are *assigned* to the physical registers in a *first phase*. In addition, *reconfiguration instructions* need to be inserted in a *second phase*, to make the physical registers accessible to the processor via the architectural registers. See Fig. 6 for an example of the resulting code.

## 3.1 Allocation of Physical Registers

This section describes the register allocation algorithm that assigns the virtual registers to the physical registers. The allocation algorithm augments the conventional register allocation of Chaitin [1] which is based on graph coloring. In addition, the algorithm aims to reduce the number of reconfiguration instructions during the second phase. Therefore, an estimate of anticipated reconfiguration costs is used, to partition the virtual registers into the physical blocks.

**Idea of the Heuristic** The basic idea of the heuristic is to assign virtual registers that are often accessed in conjunction to physical registers of the same physical block. Hence, if two virtual registers are accessed close together, and both are assigned to physical registers of the same physical block, only one reconfiguration instruction is needed to make both virtual registers accessible.

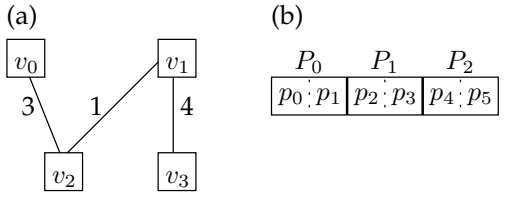


Figure 2: Example of an affinity graph.

**Affinity Graph** The decision, which virtual registers are assigned to the same physical block, is based on the number of reconfiguration instructions that are avoided by this grouping of virtual registers. The number of reconfiguration instructions, that are avoided by assigning two virtual registers to the same physical block, is expressed by the *affinity* between both virtual registers. The affinities between all pairs of virtual registers are represented in a so-called *affinity graph*. The nodes of this graph correspond to the virtual registers and each edge is weighted with the affinity between two virtual registers. The affinities are derived from the access patterns of the virtual registers. A detailed description is given below.

Figure 2a shows an example of an affinity graph for the virtual registers $v_0, \ldots, v_3$. The Figure 2b aside shows a physical register bank with 6 physical registers that are divided into 3 physical blocks. If the virtual register $v_0$ is assigned to the physical register $p_0$ and $v_2$ is assigned to $p_1$, 3 reconfiguration instructions are avoided, since the affinity between $v_0, v_2$ is 3 and they are assigned to the common physical block $P_0$.

During the allocation process, some of the virtual registers are already assigned to physical registers and thereby to physical blocks. The virtual register nodes in the affinity graph are therefore replaced by nodes representing physical blocks and the affinities are recomputed. In this way, affinities between virtual registers and physical blocks emerge that represent the number of avoided reconfiguration instructions, if the virtual register is assigned to the physical block.

So far, the interpretation of the affinity has been described. In order to *define* the affinity, the replacement strategy for register blocks needs to be discussed.

**Replacement Strategy** If a physical block must be made accessible at a program position $s$, the compiler needs to decide, which architectural block should host the physical block. The first choice is,

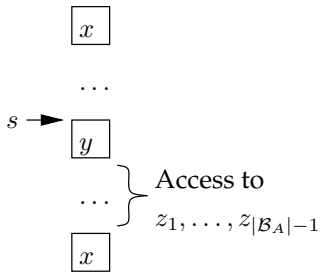


Figure 3: Replacement of physical block between two accesses.

to use an unmapped architectural block. If no such block exists, an architectural block $A$ must be chosen that is already mapped to some physical block $P$ as given by $\mu(A)$. Which block $A$ is chosen depends on its current content $P$ and is determined by the *replacement strategy.*

**Belady** The compiler uses the replacement strategy of Belady [2] that replaces the mapping of the architectural block $A$, whose physical block $P = \mu(A)$ is *latest used again.* In the following, this strategy is referred to as LUA. The strategy of Belady [2] also defines that a physical block is made accessible as late as possible (ALAP), i.e. right before the next access to the physical block.

Originally, this strategy has been presented by Belady as paging algorithm for virtual memory management and is also used for local register allocation in basic blocks.

**Persistence of Physical Blocks** For the affinity analyses it is necessary to determine, if a physical block might be replaced between two accesses to the physical block. The following theorem expresses, in which case a physical block is definitely *not* replaced between two accesses.

**Theorem 1** *According to LUA, a physical block $x$ is not replaced between two accesses to $x$, if less than $|\mathcal{B}_A|$ pairwise different blocks are accessed in between.*

**Proof 1** *This theorem is proven by contradiction. Let $x$ be replaced by $y$ at a position $s$ between two accesses to $x$, as shown in Fig. 3. Then all architectural blocks must be mapped at position $s$. Let the other architectural blocks further be mapped to $z_1, \ldots, z_{|\mathcal{B}_A|-1}$. Since $x$ is replaced, the physical blocks $z_1, \ldots, z_{|\mathcal{B}_A|-1}$ must be accessed between $s$ and the second access to $x$. All in all, with $z_1, \ldots, z_{|\mathcal{B}_A|-1}$ and $y$, $|\mathcal{B}_A|$ physical blocks need to be accessed between the two accesses to $x$, in order to replace $x$.*

### 3.1.1 Definition of Affinity

As already mentioned, the affinity between two virtual registers estimates the number of reconfiguration instructions that are avoided, if both virtual registers are assigned to the same physical block. This section gives an equivalent definition of the affinity that is used afterwards to compute affinities efficiently. The definition is based on so-called *access pairs* for two virtual registers. In the following, the symbol $v_i$ denotes an access to a virtual register $v$.

**Definition 1 (Affinity of virtual registers)** *Let $x$ and $y$ be two virtual registers. The affinity $\alpha(x, y)$ between $x$ and $y$ is defined as the number of unordered access pairs $(x_i, y_j)$ which have the following properties:*

1. *The number of pair-wise different virtual registers that are accessed between $x_i$ and $y_j$ is less than $|\mathcal{B}_A|$.*
2. *$x$ and $y$ are not accessed between $x_i$ and $y_j$.*

Let $x$ and $y$ be assigned to the same physical block $P$ and let $(x_i, y_j)$ be an access pair. Obviously, less than $|\mathcal{B}_A|$ other physical blocks are accessed between $x_i$ and $y_j$, because less than $|\mathcal{B}_A|$ virtual registers are accessed by definition. Consequently, $P$ is not replaced between $x_i$ and $y_j$ according to Theorem 1, i.e. no further reconfiguration is needed to access $y_j$. Otherwise, it *might* be necessary to insert a reconfiguration instruction between $x_i$ and $y_j$. Thus, a reconfiguration instruction is avoided for this access pair, if $x$ and $y$ are assigned to the same physical block.

The second condition in Def. 1 ensures that each avoided reconfiguration instruction is counted only once.

### 3.1.2 Construction of Affinity Graphs

In order to calculate the complete affinity graph, all access pairs are determined for each pair of virtual registers. This task can be solved for all virtual registers, by regarding for each access $v_i$, the $|\mathcal{B}_A|$ different virtual registers that are accessed after $v_i$.

Figure 4a shows an example for $|\mathcal{B}_A| = 2$ and $v_i = x_1$, in which the access pairs $(x_1, a_1)$ and $(x_1, b_1)$

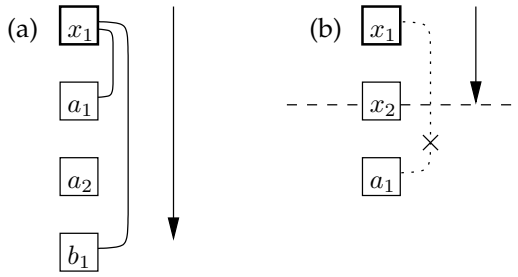


Figure 4: Calculation of affinity.

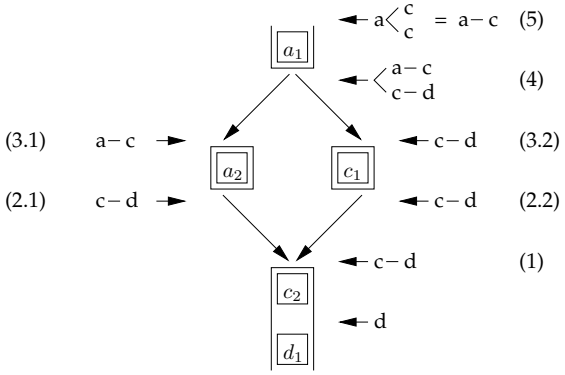


Figure 5: Example of $n$-liveness analysis.

are found, by regarding $x_1$ and the following 2 register accesses. The repeated access $a_2$ is ignored in order to ensure the second condition of Def. 1. For the same reason, all register accesses following the repeated access $x_2$ are ignored, as indicated in Fig. 4b.

To find all access pairs for a regarded register access $v_i$, the $|\mathcal{B}_A|$ virtual registers that are accessed next need to be known. For $v_i = x_1$, this would be $(a, b)$ in example 4a and $(x, a)$ in example 4b. These registers can be computed by our $n$-liveness analysis that is outlined in the following section.

### 3.1.3 Analysis of $n$-Liveness

The $n$-liveness analysis is a static program analysis that applies data flow analysis to determine, for each program position, the $n$ distinct registers that are accessed next. As already mentioned, it is used for the computation of the affinity graph. In addition, its results are used for the placement of reconfiguration instructions, as described in section 3.2.

**Trees** Figure 5 shows an exemplary control flow graph that is annotated with the results of the $n$-liveness analysis for each program position.

The results are given as trees of $n$-live registers. In the figure, the trees are drawn from left to right, i.e. the root is on the left side. The depth of the trees is limited by $n$, that is $|\mathcal{B}_A| = 2$ for the example in Fig. 5.

Position (1) is annotated with the degenerated tree `c-d`, meaning that `c` and `d` are accessed next in this very order. The same holds of course for the end of both preceding basic blocks at position (2.1) and (2.2). If virtual registers are accessed repeatedly, only the first access is considered as can be seen at position (3.2) where the access $c_2$ is ignored. At position (4), the register access sequence can not be predicted statically due to the splitting of the control flow. Therefore, the tree reflects all possible access sequences that might occur at runtime. The left access sequence contributes the tree branch `a-c`, whereas the right sequence adds the branch `c-d`. If a node has common sub-nodes, the tree is simplified like at position (5), without loosing any relevant information.

**Tree Size** Due to the limited depth and degree of each tree, the number of nodes is limited. Although the tree can theoretically consist of many nodes, it is usually rather small as basic blocks typically contain accesses to multiple virtual registers. This results in linear chains without any branches in the tree. Furthermore, the degree of a branch in the tree is limited by the degree of the corresponding control flow node. Most control flow nodes have no more than two successors.

**Computation** The $n$-live registers are computed using the general method of an iterative data flow analysis. This means that the trees of $n$-live registers are computed iteratively for the end and begin of each basic block. The convergence of the $n$-liveness analysis is not immediately obvious, because it can not be formulated as a DFA problem, which operates on a lattice. Nevertheless, we have proven the convergence of the $n$-liveness analysis in [7].

**Probabilities** To supply quantitative information about the likelihood of register accesses occurring at runtime, the nodes in these trees are annotated with probabilities. These probabilities are derived from the branching probabilities in the control flow graph. The probabilities are used as weights in several heuristics. In addition, probabilities allow us, to recognize accesses as certain, even after a divergence in control flow.

To summarize, the $n$-liveness analysis determines, for each program position, if a virtual register $v$ belongs to the $n$ virtual registers that are accessed next. This knowledge can in turn be used, to calculate the affinity graph. With the help of the affinity graph, the virtual registers are assigned to the physical registers, such that virtual registers with high affinity are assigned to the same physical block. Thus, most physical register accesses will not require a reconfiguration instruction, since the corresponding physical block is already accessible.

## 3.2 Reconfiguration

The register allocation described so far assigns the virtual registers to the physical registers of the processor. In the second phase, the reconfiguration instructions are inserted. The inserted reconfiguration instructions implicitly determine the mapping from physical to architectural registers for each program position. Hence, the physical register operands can be rewritten to refer to architectural registers.

At first, the insertion of reconfiguration instructions will be shown for a single basic block. Afterwards, this method is extended to support mappings across basic blocks.

### 3.2.1 Intra-Block Reconfiguration

The basic idea of the algorithm is to keep track of the mapping function $\mu$, while iterating over the instructions of a basic block.

At the begin of a basic block, the mapping function $\mu$ is pessimistically assumed to be undefined for all architectural blocks, i.e. $\forall A : \mu(A) = \perp$ holds.

If an instruction $i$ is found, that accesses a register of an inaccessible physical block $P$, a reconfiguration instruction is inserted directly before $i$ to make $P$ accessible. The reconfiguration instruction establishes a mapping $\mu(A) = P$, where $A$ was either unmapped or mapped to the physical block $P'$ that is latest used again.

**Example** Figure 6 shows an example of code in a basic block on the right with accesses to the physical blocks $P_0, P_1, P_2$. On the left, the current mapping function $\mu$ is indicated for a register architecture with two architectural blocks $A_0, A_1$. Since the architectural blocks are assumed to be unmapped at the begin, a reconfiguration instruction $\mu(A_0) := P_0$ needs to be inserted at the very begin of the basic block, to make $P_0$ accessible. The same applies for the second reconfiguration instruction prior to the access to

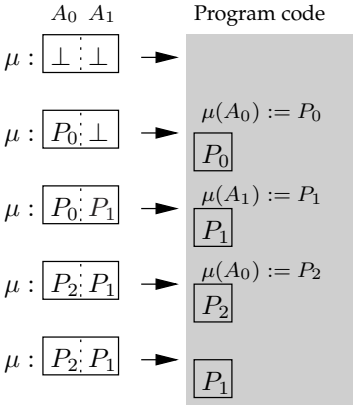


Figure 6: Insertion of reconfiguration instructions according to LUA.

$P_1$. The third reconfiguration instruction needs to replace a mapping, since both architectural blocks are already mapped to physical blocks. According to the LUA strategy the mapping $\mu(A_0) = P_0$ is replaced, as $P_0$ is latest used again.

### 3.2.2 Inter-Block Reconfiguration

Up to now, the mappings at the begin of a basic block have been assumed to be undefined. For this reason, reconfiguration instructions had to be inserted at the begin of *every* basic block.

**IN and OUT Mappings** In order to reuse the mappings from preceding basic blocks, the mappings at the begin and end of a basic block are decided in a preceding step. The mapping function for the begin of a basic block is denoted $\mu_{IN}$ and specifies, which mappings can be assumed at the begin of the basic block. Accordingly, the mapping function $\mu_{OUT}$ specifies the mappings that *must* be assured at the end of a basic block.

$\mu_{OUT} \rightarrow \mu_{IN}$ **Dependency** To assume $\mu_{IN}(A) = P$ at the begin of a basic block $b$, $\mu_{OUT}(A) = P$ must hold in all preceding basic blocks $pred(b)$. If $A$ is mapped to different physical blocks in $\mu_{OUT}$ of the predecessors, the mapping of $A$ is undefined ($\perp$) in the IN mapping. This can be expressed formally as:

$$\mu_{IN\,b} = \bigcap_{\hat{b} \in pred(b)} \mu_{OUT\,\hat{b}}$$

where $\cap$ is defined as

$$\mu_x(A) \cap \mu_y(A) = \begin{cases} P & \text{for } \mu_x(A) = \mu_y(A) = P \\ \perp & \text{else} \end{cases}$$

**Selection of Physical Blocks** The definition of $\mu_{IN}$ aims to reduce the number of reconfiguration instructions at the begin of a basic block $b$. Therefore, if $P$ is $n$-live at the begin of $b$, a mapping $\mu_{IN}(A) = P$ is arranged, if possible. The $n$-liveness guarantees that $P$ persists from the begin of the basic block up to the first access to $P$. Thus, a reconfiguration instruction is definitely avoided in $b$.

**Effect on Predecessors** To guarantee the mapping $\mu_{IN}(A) = P$, the same mapping must be defined in $\mu_{OUT}$ of all preceding basic blocks. Unfortunately, a mapping $\mu_{OUT}(A) = P$ can introduce an additional reconfiguration instruction in the predecessor. To predict, whether a reconfiguration instruction must be added to ensure $\mu_{OUT}(A) = P$, the maximum cut between the last access to $P$ and the end of the basic block is regarded. If the maximum cut is smaller than $|\mathcal{B}_A|$, the physical block $P$ will persists until the end of the preceding basic block according to LUA. Hence, this case requires no additional reconfiguration instruction in the predecessor.

**Estimation of Costs** The disadvantage of an additional reconfiguration instruction in a predecessor is expressed in terms of *costs*. The costs of a reconfiguration instruction in a block $b$ is given by the execution frequency of the basic block. Hence, the costs for a mapping $\mu_{IN}(A) = P$ is given by the sum of costs of additional reconfiguration instructions in the preceding basic blocks. If these costs are lower than the costs of a reconfiguration in the basic block, the mapping will be defined. As a result, reconfiguration instructions are moved out of loops into a block preceding the loop.

### 3.3 Extensions for Multi-Core Processors

So far, the register allocation has been discussed for a single-processor. This section outlines how the register allocation can be extended for processors with multiple synchronous cores, reconsidering only few aspects. The processor is assumed to have the structure shown in Fig. 7.

It consists of a shared physical register file that can be accessed uniformly by every core. Each core has its own set of architectural registers to establish its mapping independently.

Since the cores access a shared register file, the creation of the conflict graph and the coloring treat parallel instructions like a single VLIW instruction,

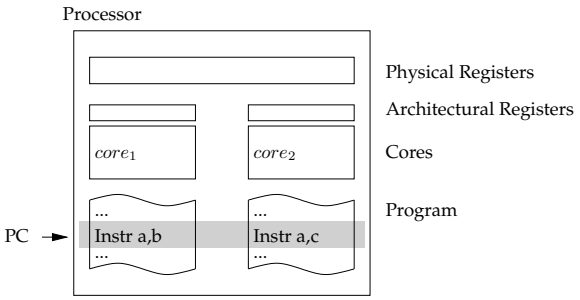


Figure 7: Multi-core processor architecture.

whereby the conventional algorithms for single processors can be applied.

The affinity analysis and the insertion of reconfiguration instructions are performed for each core separately, since each core has a dedicated architectural register bank.

## 4 Evaluation

This section compares the efficiency of a reconfigurable register bank to static register architectures, using several benchmark programs.

First, the tool chain of the evaluation and the benchmarks is outlined. Afterwards, several aspects of the reconfigurable register architecture are analyzed.

**Tool Chain** The register allocation algorithm has been implemented in an optimizing compiler for a streamlined RISC processor called S-Core. The processor is described in [8] and was developed in the GigaNetIC project. The compiler augments the lcc [9] frontend with a complete suite of global optimizations and conducts global register allocation [1]. For details about the compiler, refer to [10].

The benchmarks were evaluated using a cycle accurate simulator. Most instructions take 1 cycle on the S-Core RISC processor. Memory read and write instructions have been assumed to take 3 cycles on average (cache hit: 2 cycles; cache miss: 5 cycles) Reconfiguration instructions have been counted as 2 cycles. A recent virtual hardware prototype has shown that reconfiguration should be feasible even in one cycle.

**Benchmarks** In order to utilize the 32 registers of the reconfigurable register bank, benchmarks with an inherent high register pressure have been chosen. Nevertheless, high register pressure also results

from preceding compiler optimizations, like "common subexpression elimination", "register coalescing", and certain scheduling strategies, like "software pipelining".

For each benchmark, the maximum cut of the virtual register's life ranges is shown in Table 1. The maximum cut is a lower bound for the number of registers that are required to avoid all spill code.

In the following, a short description of each benchmark is given

**convolution** computes the discrete convolution of a 100 element array with a 32 element array. The results are written into another 100 element array.

**fftlike** simulates the variable access pattern of a fast fourier transformation. The intermediate results are alternately stored in two arrays of 16 elements.

**median** computes the median of each $n$ consecutive elements in an array of 100 elements. The computation is executed simultaneously for $n = 1, 2, 3, \ldots, 32$ in a single pass.

**mm4** multiplies two $4 \times 4$ matrices. The result is again stored in a $4 \times 4$ matrix

**poly** evaluates a polynomial of degree 32 with variable coefficients. The evaluation is repeated 100 times.

**Register Architecture** Figure 8 shows the number of simulation cycles for a normal and a reconfigurable register architecture. In the case of the normal register architecture, the processor accesses the registers directly. For the reconfigurable register architecture, the processor accesses 32 physical registers via 16 architectural registers. Thus, the instruction set is the same in both cases. Furthermore, the architectural registers are divided into 4 blocks of 4 registers each in case of the reconfigurable architecture.

Since the benchmarks have a high register pressure, they benefit from the additional 16 registers that are offered by the reconfigurable architecture. Table 1 lists the exact number of execution cycles and the number of inserted reconfiguration instructions. It is remarkable that the number of simulation cycles decreases by about 30% percent for the reconfigurable architecture.

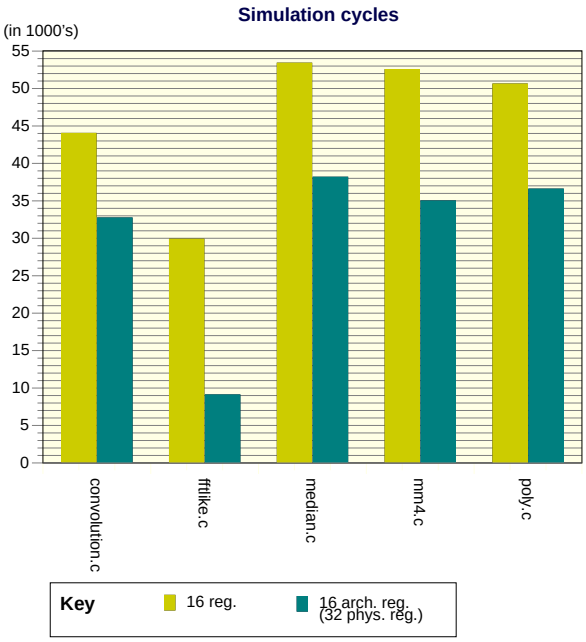


Figure 8: Simulation results for different register architectures.

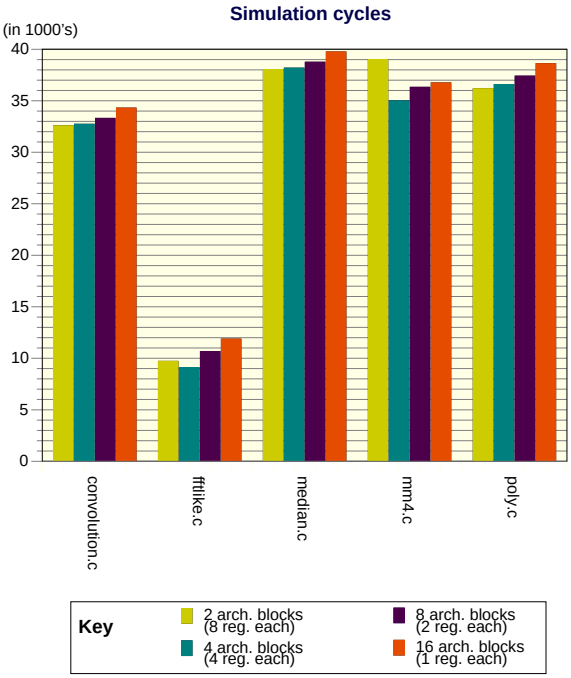


Figure 9: Simulation results for different block sizes.

**Block Size** In the previous benchmark, the architectural registers have been divided into 4 blocks. In this section, the influence of the number of blocks is analyzed for a fixed number of 16 architectural and

| Benchmark | Maxcut | Cycles (16 regs.) | Cycles (32 reconf. regs.) | Reconf. instr. | Total instr. |
|---|---|---|---|---|---|
| convolution | 32 | 44078 | 32786 | 22 | 260 |
| fftlike | 35 | 29958 | 9169 | 36 | 205 |
| median | 33 | 53450 | 38232 | 13 | 329 |
| mm4 | 44 | 52582 | 35054 | 34 | 285 |
| poly | 22 | 50660 | 36646 | 9 | 83 |

Table 1: Benchmark characteristics and results.

32 physical registers.

For the benchmarks `poly`, `convolution` and `median`, the number of simulation cycles increases with a growing number of architectural blocks, compare Fig. 9. This behavior is caused by the access pattern of the benchmarks. The registers are accessed repeatedly in the same order. Hence, at least $|\mathcal{B}_A|$ reconfiguration instructions must be inserted, to switch between the 32 physical registers.

In contrast to that, the matrix multiplication benchmark accesses rows and columns of 4 registers repeatedly. Thus, a block size of 4 architectural registers performs best for this benchmark.

For larger programs smaller block sizes require more reconfiguration instructions to establish well-defined mappings, for example around each function call. Also, smaller block sizes limit the effect of each reconfiguration instruction to fewer register accesses. Kiyohara [5] constitutes an extreme case with one reconfiguration instruction per access to an extended register and a fixed set of core registers, which can be accessed without any reconfiguration.

Our approach provides a similar distinction, based on the access frequency of a register. If a register like the stack pointer is accessed frequently, its block will not be unmapped by our register allocation.

**Affinity** As mentioned in section 3.1, the affinity analysis intends, to reduce the number of reconfiguration instructions.

Figure 10 shows the actual effect of the affinity analysis, compared to a random assignment of virtual registers to physical register blocks. The number of simulation cycles decreases by up to 50% due to reduced number of reconfiguration instructions.

**Code Density** For our benchmark, reconfiguration instructions increase code size by about 12% on average. This result is based on an architecture with 16 architectural and 32 physical registers. Instructions are encoded in 16 bits and support at most two register operands. Thus, an ideal extension of this

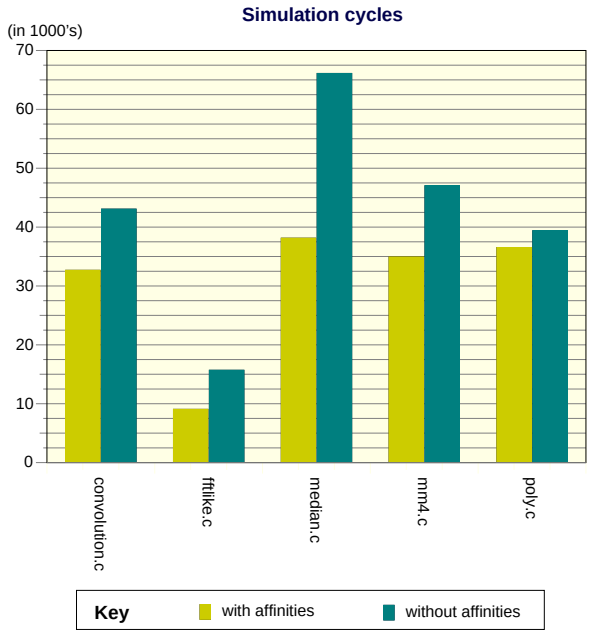


Figure 10: Effects of the affinity analysis.

architecture to 32 freely addressable registers would require instructions of 18 bits minimum. This results in code growth of 12.5% or more.

Realistically, the instruction format would have to be extended to 24 or even 32 bits, which cuts code density in half and leaves a huge share of the opcode space unused. In contrast, reconfigurable architectures constitute a more flexible means to add more registers, while preserving the overall instruction format.

# 5 Conclusion

A reconfigurable register architecture offers additional registers to the executing program, at additional cost for reconfiguration code. The instruction format remains unchanged. Programs with high inherent register pressure, and compiler optimizations,

which increase register pressure, do benefit from these additional registers.

We have described an augmented register allocation phase for a compiler, which utilizes all physical registers and manages their mapping to architectural registers automatically. Static program analyses are used to exploit the block structure of the reconfigurable register bank and to reduce reconfiguration overhead at runtime.

The evaluation of the register block size has shown, that reasonably sized register blocks yield a modest reconfiguration overhead. Large register blocks suit especially repeated accesses to many registers. For programs with irregular access sequences, smaller block sizes yield slightly better results.

Our affinity analysis further reduces the number of reconfiguration instructions. For programs with high register need, offering two physical registers per architectural register (instead of one) yields a speedup of about 30%.

# References


[1] Chaitin, G.J.: Register allocation & spilling via graph coloring. In: SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction, New York, NY, USA, ACM Press (1982) 98–101

[2] Belady, L.A.: A study of replacement algorithms for virtual-storage computer. IBM Systems Journal **5**(2) (1966) 78–101

[3] Tomasulo, R.: An efficient algorithm for exploiting multiple arithmetic units. IBM J. Res. Dev. **11**(1) (1967) 25–33

[4] SPARC International: The SPARC architecture manual: Version 8. Prentice-Hall, Upper Saddle River, NJ 07458, USA (1992)

[5] Kiyohara, T., Mahlke, S., Chen, W., Bringmann, R., Hank, R., Anik, S., Hwu, W.M.: Register Connection: A New Approach to Adding Registers into Instruction Set Architectures. In: ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture, New York, NY, USA, ACM Press (1993) 247–256

[6] Ravindran, R.A., Senger, R.M., Marsman, E.D., Dasika, G.S., Guthaus, M.R., Mahlke, S.A., Brown, R.B.: Increasing the number of effective registers in a low-power processor using a windowed register file. In: CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems, New York, NY, USA, ACM Press (2003) 125–136

[7] Dreesen, R.: Registerzuteilung für Prozessor-Cluster mit dynamisch rekonfigurierbaren Registerbänken. Diploma-Thesis, University of Paderborn (2006)

[8] Langen, D., Niemann, J.C., Porrmann, M., Kalte, H., Rückert, U.: Implementation of a RISC Processor Core for SoC Designs - FPGA Prototype vs. ASIC Implementation. In: Proceedings of the IEEE-Workshop: Heterogeneous reconfigurable Systems on Chip (SoC), Hamburg, Germany (2002)

[9] Fraser, C.W., Hanson, D.R.: A Retargetable C Compiler: Design and Implementation. Benjamin/Cummings Pub. Co., Redwood City, CA, USA (1995)

[10] Bonorden, O., Brüls, N., Le, D.K., Kastens, U., Meyer auf der Heide, F., Niemann, J.C., Porrmann, M., Rückert, U., Slowik, A., Thies, M.: A holistic methodology for network processor design. In: Proceedings of the Workshop on High-Speed Local Networks held in conjunction with the 28th Annual IEEE Conference on Local Computer Networks (LCN2003). (2003) 583–592