Dynamic Register Allocation for Multi-Threaded Network Processors

Ryan Collins
Georgia Institute of Technology
College of Computing
Atlanta, GA, 30332-0280
rcollins@cc.gatech.edu

Fernando Alegre
Georgia Institute of Technology
College of Computing
Atlanta, GA, 30332-0280
fernando@cc.gatech.edu

Xiaotong Zhuang
Georgia Institute of Technology
College of Computing
Atlanta, GA, 30332-0280
xt2000@cc.gatech.edu

Santosh Pande
Georgia Institute of Technology
College of Computing
Atlanta, GA, 30332-0280
santosh@cc.gatech.edu

October 12, 2005

Abstract

Modern network processors such as the Intel IXP family hide the latency of slow instructions by supporting multiple threads of execution. Context switches in the IXP architecture are designed to be very fast. However, the low overhead is partly achieved by leaving register management to programs, with little support from the hardware. The complexity of the multi-engine, multi-threaded environment makes manual register management a daunting task, which is better left to the compiler. However, a purely static analysis may not be able to achieve full utilization of the register file due to conservative estimates of liveness. A register

that is live across a context switch point must be considered live for the duration of all other threads, and so it must be assumed to be unavailable to other threads. In addition, aliasing further reduces the effectiveness of static analysis. The net effect is a large number of idle cycles that are still present after static optimization.

We propose a dynamic solution that requires minimal software and hardware support. On the software side, we take a pre-allocated binary file and annotate the potential context switch instructions with information about the dead registers. On the hardware side, we try to rename all transfer registers and addresses to dead general purpose registers and update the vector of used registers. We then replace the long-latency memory instructions with fast move instructions in the architecture using the dynamic context. The results show up to 51% reduction in idle cycles and up to 14% increase in the throughput for hand coded applications.

1 Introduction

The current demands of Internet traffic have created a need for fast, programmable network processors. The need for flexibility arises as the complexity and diversity of network tasks increases. Current tasks range from IP forwarding to computing the MD4 hash of a packet to scanning packet contents for potentially malignant code. Packet processing must also keep up with increasing network speeds. On the OC-768, the processor must complete packet processing in only 13ns to avoid packet loss.

Typically, a network processor has multiple microengines on-chip and hardware support for multiple threads on each engine to allow work to be done in parallel. Each microengine can process a separate packet in parallel. The threads are used to mask the latency of long memory operations, which are frequent due to the lack of cache. Upon executing a long-latency operation, the processor may also execute a context switch (if the programmer explicitly encodes one in the long-latency instruction) to hide the latency of the operation. This work is done entirely by the programmer or compiler; there is no OS to schedule the threads and prevent starvation.

Network processors also contain a large number of registers to decrease the number of memory accesses. Currently, programmers write the vast majority of network application code in the processor's native assembly language or a low-level restricted version of C. There have been efforts to construct an optimizing compiler with support for high level languages, but this work is still in progress [9] [6]. There are too many registers for the programmer to effectively allocate by hand, so a small compiler or allocator must determine how to best divide the registers across threads. In the simplest case, the compiler evenly divides the register set across each thread and does not perform any inter-thread analysis. In most cases, identical programs are executing on each thread, so this method produces acceptable results. [11] proposes an alternate static algorithm which balances register allocation across threads according to their needs. This results in a performance gain for both *Symmetric Register Allocation* (SRA) (one in

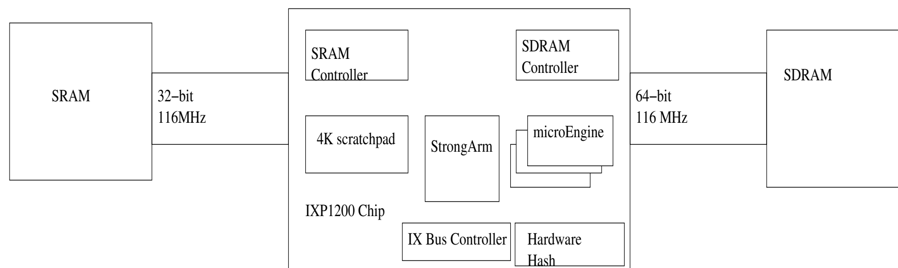

Figure 1: The IXP Block Diagram

which all threads execute the same code) and *Asymmetric Register Allocation* (ARA) (one in which each thread executes different code) schemes. However, due to static nature of allocation, the assumptions used by this inter-thread register allocation are conservative; dynamically, more aggressive opportunities exist for saving memory traffic and latencies by undertaking register allocation during execution. By effectively utilizing the information about the dynamic context one can eliminate spills as well as aliased memory load/stores leading to reduction in idle cycles and increase in the throughput which is the theme of the paper.

2 The Intel IXP1200

The rest of this paper will use the Intel IXP1200 (cf. figure 1) processor as the target for the optimizations [1]. This processor consists of multiple RISC cores connected by a common bus, so that they can either work in parallel or in a pipeline. This section discusses some of the key features that factor into this paper and shows the format of the corresponding assembly instructions.

2.1 Arithmetic and Logic Unit

Most IXP-architecture assembly instructions have a common pattern:

```
unit[arg1,arg2,...] options
```

where `unit` is one of the hardware units that compose the processor. In this paper, we are basically concerned with `alu` instructions, which operate on general purpose registers, and with memory instructions, discussed below.

The first argument indicates the destination operand, which may be either a general-purpose register from one of the two banks available (known as A and B) or a special transfer register from one of the two sets (known as SRAM-XFER and SDRAM-XFER) connected to the memory. The source operands may also

[1] The main reason for using IXP 1200 was the availability of the simulator; it may be noted that for the scope of this work, IXP 2400/2800 offer the same problem

be in one of those four sets, but each source operand must be in a different set. The ALU is used not only for arithmetic and logic operations, but also for bit shifts and copying data between registers.

General-purpose registers can be addressed either globally or locally. The former is denoted by using the prefix `@` while the latter uses no prefix. In both cases, the register is denoted by the bank (either A or B) followed by a number. Relative addressing means that registers are partitioned in equal subsets, each assigned to a thread. For example, if there are 16 A registers per thread, then thread 0 will use registers `@A0` to `@A15` and thread 1 will use registers `@A16` to `@A31`. Thus, a reference to `A1` will actually use `@A1` when executed by thread 0 and `@A17` when executed by thread 1. Since code is often shared by threads, this means that each reference to a relative register uses actually as many registers as threads.

2.2 Exposed Memory Hierarchy

Programmers are required to determine where to store a particular piece of data, since there is nothing equivalent to the transparent cache hierarchy found in modern general-purpose computers.

The IXP architecture provides for at least two types of memory: `sram` and `sdram`. Data is copied between a transfer register and the corresponding memory controller asynchronously, and the processor is signalled when the transfer is finished. Programs typically perform a context switch so that they swap out a thread waiting for a memory transfer and swap in a thread ready to perform ALU operations, so that the effect of memory latency is diminished.

Transfer registers are the only way to move data between the ALU and the memory. They are denoted by a prefix ($ for sram,$$ for sdram) followed by a number. There are actually two registers associated to each number: a read-only register for transfers from memory to the ALU, and a write-only register for transfers in the other direction. Therefore, data moved to a write-only register cannot be read back.

For example, the instruction

```
sram[write, $0, a1, 0, 1]
```

stores the contents of the sram-xfer register (`$0`) to the SRAM memory address stored in general-purpose register `A1`. This instruction typically has a latency of 20 cycles.

On the other hand, the instruction

```
sdram[write, $$1, b2, 0, 1]
```

stores the contents of the sdram-xfer register (`$$0`) to the SDRAM memory address stored in register `B2`. This instruction typically has a latency of 50 cycles.

It is not possible to move data directly between a general-purpose register and memory. Thus, the instruction

```
sram[write, a0, a1, 0, 1]
```

is illegal and needs to be replaced by a pair of instructions:

```
alu[$0, --, b, a0]        // copy from A0 to sram-xfer 0
sram[write, $0, a1, 0, 1] // copy from xfer register to memory
```

Similarly, instead of the following two instructions

```
sram[read, a0, a1, 0, 1]   // [illegal] read directly into GPR
alu[a0, a0, +, b0]         // srcs GPR,GPR and dest GPR
```

we must write these:

```
sram[read, $0, a1, 0, 1]   // read into xfer register
alu[a0, $0, +, b0]         // srcs GPR,xfer and dest GPR
```

2.3 Fast Context Switches

Context switches in the IXP architecture save only the current PC, which is then replaced by the PC of the next thread. Since no other register is saved, a context switch can be completed in a single cycle. Furthermore, context switches must be explicitely requested by the programmer by using the keyword `ctx_arb` either by itself or as an option to a memory transfer instruction.

Let us consider the following example:

```
L0: immed[@a0, 5]         // global register A0 is set to 5
L1: ctx_arb               // switch context
L2: alu[a1, --, b, @a0]   // copy global A0 to thread-relative A1
...
L4: immed[@a0, 6]         // global register A0 is set to 6
L5: ctx_arb               // switch context
L6: alu[a1, --, b, @a0]   // copy global A0 to thread-relative A1
L7: ctx_arb               // switch context
```

Suppose thread 0 is at `L0` and thread 1 is at `L4`, and thread 0 is executing. Then global `A0` will be set to 5 and execution will jump to `L4`, which will set `A0` to 6. Then, context will switch back to thread 0 at `L2`, and thus, local register `A1` will be set to 6.

On the other hand, if thread 1 was initially at `L6`, then local register `A1` (which is global `A9` when there are 8 A registers per thread) will be set to 5.

This illustrates that using global registers produces results that depend on the current execution context, and which may not be what was expected. The programmer can avoid this confusion by using context-sensitive register references. Replacing `@A0` with its local counterpart `A0` will produce the expected result at the expense of using a total of 4 actual registers instead of 3.

Benchmark	*# Idle Cycles*	*% Total Cycles*
ipfdwr (1 ME)	71	0%
ipfdwr (4 ME)	88210	0.184%
md4 (1 ME)	620388	2.58%
md4 (4 ME)	9574894	19.95%
nat (1 ME)	101284	0.422%
nat (4 ME)	106866	0.223%
url (1 ME)	1006534	4.19%
url (4 ME)	7728240	16.10%

Table 1: # of Idle Cycles for each Benchmark

3 Motivation

Since network processors have real-time constraints, it is critical that they do not waste cycles running `nop` instructions. However, the lack of a cache, the high cost of memory accesses, and the symmetric programs typically executed on the IXP create a situation where long periods of idle activity are inevitable. In the symmetric programming style, it is likely that each thread will reach a given memory instruction at the same time. The first thread executes memory operation and passes control to the second thread which executes a memory operation and passes control to the third thread and so on. Since each memory operation requires a large number of cycles, the processor will enter a stage where all four threads are waiting for their memory operations to complete, and the only option is to begin issuing `nop` instructions. Some experiments were performed to verify this conjecture. Table 1 shows the number of cycles that each benchmark spends in the idle state. One can see that for multi-threaded cases, there is a large number of idle cycles spent by a given micro-engine.

Clearly, reducing the number of memory accesses will increase efficiency. A way to reduce memory access is by transforming some memory references into register references. Table 2 shows that the register utilization for most benchmark is actually quite low. The register utilization can be computed using the following formula,

$$\frac{\sum_{r \in regset}^{numRegs} \sum_{i=0}^{numCycles} liveAt(r,i)}{numRegs * numCycles}$$

The above formula determines how long a register is occupied (in terms of cycles) on an average during the program execution. If the hardware could divert some memory traffic to the unused registers, the number of idle cycles would be reduced.

[11] describes an algorithm that statically partitions the register file into shared registers and private registers. Shared registers can be accessed by all threads safely, while private registers can only be accessed by one particular thread. Conservatively, shared registers must not be live across any context

Benchmark	*Register Utilization*
ipfdwr (1 ME)	22.27%
ipfdwr (4 ME)	23.02%
md4 (1 ME)	17.94%
md4 (4 ME)	12.67%
nat (1 ME)	25.74%
nat (4 ME)	25.35%
url (1 ME)	12.95%
url (4 ME)	10.60%

Table 2: Register Utilization Pre-Optimization

switch point. Even though this algorithm results in a large speedup over the traditional network processor technique of partitioning the register file into equal sets of private registers, it cannot fully utilize the register set because of the conservative assumption that the other threads may be executing any possible instruction in their context. Since the technique is purely static, it has to assume all possible orderings of thread executions and thus the technique assumes that a given register to a thread would be busy throughout the execution duration of another thread. In short, it can not aggressively allocate private registers.

Let us consider the following example:

```
    ...                   // no references to @b15 or b15
    ...                   // above this point
L0: br!=ctx[0,L1]         // jump to L1, unless we are thread 0
    alu[@b15,a16,+,b17]   // b[15]=a[ctx*Anum+16]+b[ctx*Bnum+17]
    ...                   // more code, but no branching
    ctx_arb               // ctx=next(ctx); goto pc[ctx];
    ...                   // more code, but no branching
    alu[b17,a16,+,@b15]   // b[ctx*Bnum+17]=a[ctx*Anum+16]+b[15]
L1: ...                   // no references to @b15 or b15
    ...                   // below this point
```

The register `@B15` is live across a context-switch point. Assume each thread executes the code at `L0` once. If there were just one thread, then static analysis would allow us to conclude that `@B15` is dead at `L1`, and so it can be reused after execution reaches that point. However, in the presence of several simultaneous threads executing at different points in the code, we must use some dynamic mechanism to check that all threads have reached `L1` before we can declare `@B15` dead.

Increased register use would allow for reduction of redundant memory accesses. Table 3 shows that a large portion of memory accesses are comprised of redundant loads. The double-load column shows the number of times the program loads a value from memory twice without storing anything to that location in between. Common causes are register spills, infrequent uses of values

Benchmark	*# ME*	*Double Loads*	
		Total	*% Mem Accesses*
ipfwdr	1	349931	64.5%
ipfwdr	4	1319746	64.1%
md4	1	115367	11.1%
md4	4	152655	6.47%
nat	1	373967	58.3%
nat	4	548310	55.0%
url	1	479226	49.5%

Table 3: Memory Access Patterns for handwritten benchmarks

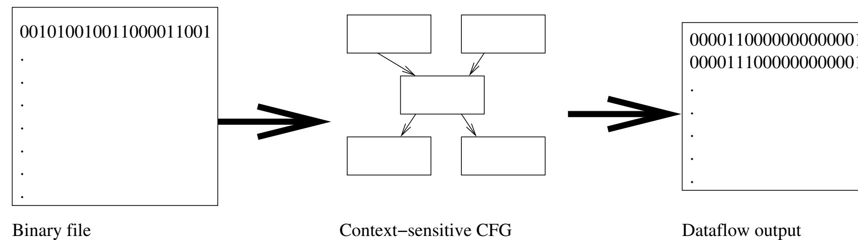

Figure 2: Overview of static side

over long lifetimes and aliasing. Obviously, diverting some of these accesses to unused registers would reduce memory latency.

In conclusion, a significant opportunity exists to reduce the load/stores to the memory and idle cycles. Faster thread execution leads to higher throughput, which is the main purpose of network processors. This paper presents a combination of static and dynamic mechanisms to put dead registers to work towards that goal. The algorithm consists of two parts. First, static analysis finds register usage patterns, and then that information is used dynamically to map memory addresses to dead registers.

4 Static Implementation

The algorithm takes a binary file as input and outputs an annotated version of the file (cf. figure 2.) It is assumed that the existing register allocator uses a simple allocation strategy that divides the register set evenly among the threads. However, the optimization will still work in the presence of a more advanced allocation strategy.

First, the algorithm creates a control flow graph (CFG) that divides the blocks into non-switch regions [11]. Non-switch regions are basic blocks that have been sub-partitioned to include at most one context-switch instruction at

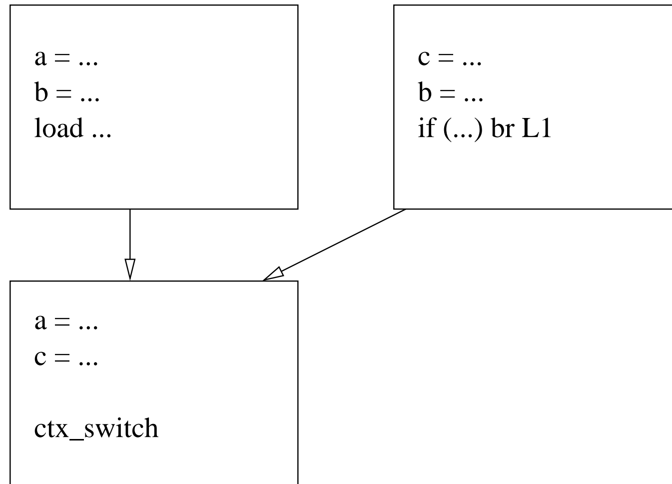

Figure 3: A non-switch region CFG

its boundaries, just as normal basic blocks contain at most one branch instruction. Figure 3 shows an example of a CFG that has been divided into non-switch regions.

Next the scheme discovers dead-until-end registers by undertaking analysis. The algorithm uses the following dataflow equations to discover the dead registers,

$$
\begin{aligned}
\text{Dead-Out}(BB) &= \textstyle\bigcap_{s \in \text{Succ}(BB)} \text{Dead-In}(s) \\
\text{Dead-In}(BB) &= \text{Dead-Out}(BB) - \text{Use}(BB) - \text{Def}(BB) \\
\text{Dead-In}(EXIT) &= U
\end{aligned}
$$

It is important to note that `Dead-In` and `Dead-Out` are not the inverse of the traditional `Live-In` and `Live-Out`. A dead register is made live by a use or a def. This is because it is unsafe to use a dead register across a new definition of it, even if that definition is never used. For example,

```
alu[a14, a15, +, b15]
alu[a15, a14, +, b15]
...
alu[a14, a15, +, b15]
...
```

By traditional dataflow analysis, `a14` is dead after its use in the second line. However, it's cleary unsafe to treat `a14` as if it is dead for the rest of the program.

This work only finds registers that are dead for the duration of the program. Since aliasing allows multiple names to map to the same address, it is unsafe to unmap an address before the end of the program. In general, we also found that globals are the most heavily used aliased variables which are live through the execution of the program and we benefit most for their run time allocation.

The dead registers at several program points are collected into a hash-table. At program initialization time, this table is loaded into SRAM. The bit vector is 128 bits long. If the program uses only thread relative register references, then the bit-vector size can be reduced to 32 bits.

Currently, the algorithm does not handle the case where the program uses memory operations to communicate between microengines. This case cannot be handled efficiently on the IXP1200, which the lacks the IXP2800's Next Neighbor register communication mechanism [5]. So, first the algorithm determines which addresses are used for inter-engine communication and which are used for spill values. Then, it extends the option field of every memory operation to note whether a memory operation is local or global.

5 Dynamic Implementation

The goal of dynamic part of this work is to map memory addresses to dead registers. The hardware then replaces any instructions which use the mapped memory address with fast register-move instructions. The registers identified to be dead by static phase above (and loaded through the SRAM hash table) are used for allocation. For example, if the address 128 is the target of a store instruction and GPRs A15-17 are dead, it can rewrite the following instruction sequence,

```
alu[$0, a3, +, b4]
sram[write, $0, 128, 0, 1] CTX_SWAP
...
sram[read, $0, 128, 0, 1], CTX_SWAP
alu[a1, b2, +, $0]
```

as

```
alu[a16, a3, +, b4]
alu[a15, --, b, a16]
...
alu[a17, --, b, a15]
alu[a1, b2, +, a17]
```

Now suppose that only GPR A15 is dead. The hardware greedily allocates the dead register to next address that requires it, the write transfer register $0. If there are no remaining registers to allocate for the address 128. The following instruction sequence will be generated by the hardware: [2]

[2] We used symbolic machine-code notation with mnemonics similar to assembly instructions instead of actual machine codes

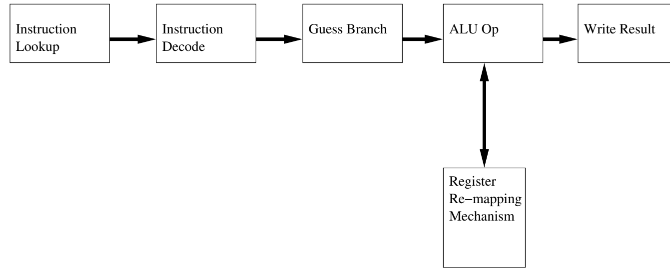

Figure 4: The modified IXP pipeline stages

Register	*Type (4 bits)*	*Mem Address (17 bits)*	*Register Map (7 bits)*
a3	0	0x0F000	b4

Table 4: Example Book-keeping Entry

```
alu[a15,a3,+,b4]
alu[$0, --,b,a15]
sram[write,$0,128,0,1],ctx_swap
...
sram[read,$0,128,0,1],ctx_swap
alu[a1,b2,+,$0]
```

This transformation preserves the correctness of the original code, at the cost of an additional move instruction.

The new hardware to perform the above dynamic allocation is inserted into the fourth pipeline stage. (see Figure 4). At that point, the ALU output forms the memory address for any memory operation. This allows the hardware to use the actual value of a memory address instead of an alias for it.

Figure 5 formally illustrates the finite state machine for the allocation hardware. The hardware checks if the current memory address is already stored in the table; if so, it replaces the memory operation with a register move instruction. If the current memory address is not in the table, it allocates one of the remaining dead registers and creates a map between the dead register and the address. The remainder of this section explains in detail how this process is done in hardware.

The new storage hardware consists of a CAM table and a hash table. The 8-entry CAM table is a mapping between a memory address and a register which contains the contents of the memory address. The 64-entry hash table is a mapping between a program point (PC) and the dead registers available at that program point.

Each book-keeping entry contains the type of memory acccessed along with the address. The type tag also doubles as a valid bit, with type 0 corresponding

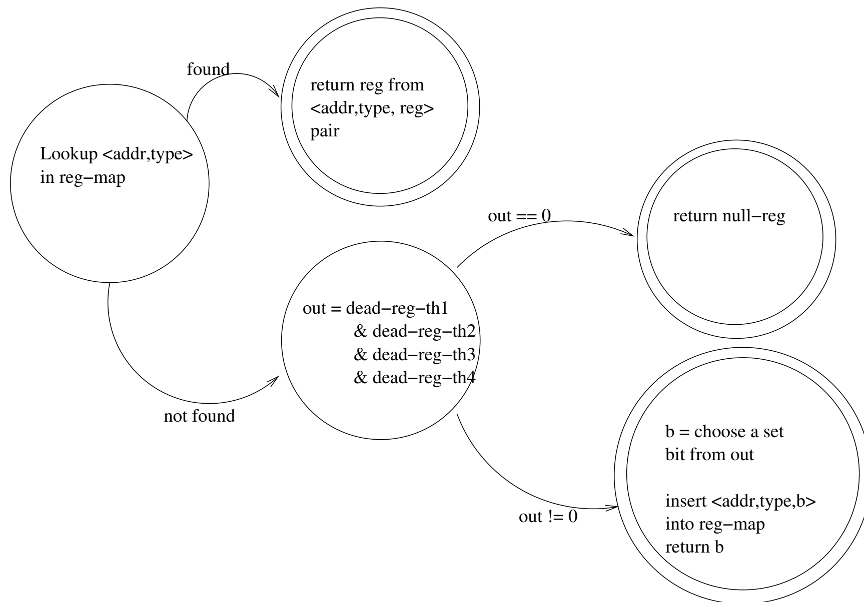

Figure 5: Dynamic Allocation FSM

to the invalid state. The memory is then bypassed by accessing the register in the Register Map entry. In the case of table 4, register `@b4` contains the contents of memory address 0x0F000.

If the current address is not in the lookup table, the hardware allocates an unused register. It determines that a register is a candidate for allocation if it appears in the intersection of the dead-rest bit-vectors for all four threads at the most recent context switch. For example, if registers 0-2 are dead in threads 0-2 and registers 0,2 are dead in thread 3, the hardware computes

```
1110...0 & 1110...0 & 1110...0 & 1010...0 = 1010...0
```

so registers 0 and 2 are candidates for allocation. It arbitrarily selects the first bit which is set, so in this case the hardware will allocate register 0 to the next memory address.

Table 5 shows the contents of the 64-entry hash table. The 10-bit PC is the index into the hash table, but the hardware only stores 5-bits for the PC in the table. The software side generates a minimal perfect hash function [1] that maps each 10-bit PC onto a 5-bit number. Since the program only uses a small subset of the entire PC range as dead-register points, the software can generate a fast hash function that correctly maps a 64-entry subset of program points onto 5-bit numbers. The 32-bit dead-register bit-vector contains a listing of all (thread-relative) dead-registers at the given program point. If the program uses absolute register references in addition to thread-relative register references, we

PC Hash (5 bits)	*Dead-Register Bit-Vector (32 bits)*
0	0010...0
⋮	⋮
63	0110...1

Table 5: The 64-entry hash table

must keep a 128-bit dead-register bit-vector which increases the storage space by a large amount.

The new hardware affects the latency of all memory operations. By default, even if the hardware cannot create a mapping between the memory operation and a register, there is still a latency penalty of 2 cycles for the opcode match followed by the CAM lookup. If there is a match for the memory address in the CAM table, the operation must take an additional 5 cycle latency penalty to restart the pipeline with the new move instruction that replaces the old memory operation. Finally, if there is no match for the memory address, but there exist free dead-registers, there is another 1 cycle of latency for the test-and-set operation to allocate a dead-register. So, the latency ranges from 2 cycles (default case) to 8 cycles (unmatched memory address + free dead regs). The hash-table lookup is performed in parallel with the CAM lookup, so it does not add any extra cycles to the overall latency penalty.

The overall chip size increases by 323 bytes. The 8-entry CAM table requires 27 bits $*$ 8 entries = 27 bytes. The 64-entry PC hash table requires 37 bits $*$ 64 entries = 296 bytes.

It may be noted that the solution is off the critical path which does not affect the clocking speed or the latency involved in the design. First, such a processing is only done at context switch points caused by memory (SRAM or SDRAM) accesses. Prefetching I used to get the relevant entry of the mapping table (due to space limitations we can not get into the details of this part but in short when a non-context switch region is entered all its exit entries are prefetched). Thirdly, the memory access is not delayed beyond the corresponding latency since these operations are performed in parallel. The only thing that gets delayed is the decision to context switch. In most cases, when dead registers are found, the context switch does not happen letting the current thread to continue, only when dead registers are not found, one context switches the added latency here being 8 cycles.

We now present a full example to illustrate the technique.

Example. Suppose 4 registers in the A bank and 4 registers in the B bank are available. Given the following code sequence:

```
L0: alu[a3, a1, +, b2]
alu[b2, a0, +, b0]
alu[$0, a3, +, b2]
```

```
alu[a3, a1, +, b2]
alu[b2, a0, +, b0]
alu[$0, a3, +, b2]
sram[write, $0, 128, 0, 1], ctx_swap
```

dead−regs: 11111111111111111111111111111111

```
alu[a3, a2, +, b2]
sram[read, $0, 128, 0, 1], ctx_swap
```

dead−regs: 11111111111111111111111111111111

```
alu[a1, b2, +, $0]
```

dead−regs: 11111111111111111111111111111111

```
bbend
```

dead−regs: 11111111111111111111111111111111

Figure 6: Example 1 - The Non-Switch Region CFG

```
L1: br!=ctx[0, L2]
L2: sram[write, $0, 128, 0, 1], ctx_swap
alu[a3, a2, +, b2]
sram[read, $0, 128, 0, 1], ctx_swap
L3: alu[a1, b2, +, $0]
```

Figure 6 shows the example as divided into non-switch regions. Then, figure 7 shows the example after dataflow analysis has been performed. Before L0, the following registers are dead: b1,b3. Before L2, the following registers are dead: a0,b0,b1,b3.

Now when the hardware reaches the third instruction, it must replace the register $0 with a general purpose register. The updated 8-entry CAM table is shown in table 6 So, the new instruction sequence is:

```
L0: alu[a3, a1, +, b2]
alu[b2, a0, +, b0]
alu[b1, a3, +, b2]
L1: br!=ctx[0, L2]
L2: sram[write, $0, 128, 0, 1], ctx_swap
alu[a3, a2, +, b2]
sram[read, $0, 128, 0, 1], ctx_swap
L3: alu[a1, b2, +, $0]
```

The dynamic register allocator changes the fifth instruction to a new move instruction. The new contents of the CAM table are shown in table 7.

```
L0: alu[a3, a1, +, b2]
alu[b2, a0, +, b0]
```

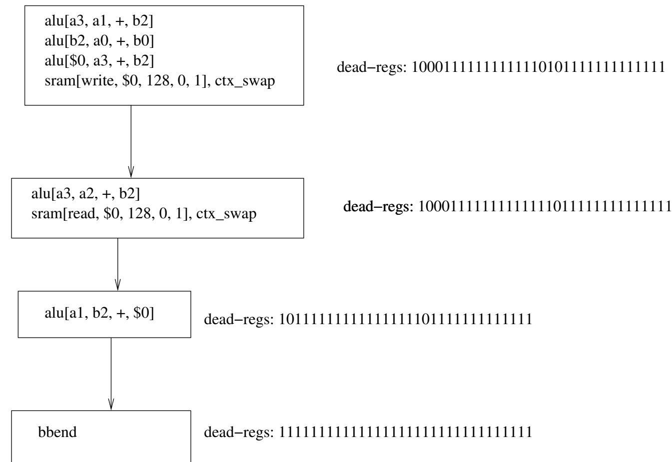

Figure 7: Example 1 - The Dead Registers found by DFA

Type (3 bits)	*Mem Address (17 bits)*	*Register Map (7 bits)*
sram-wr-xfer	0	b1
–	–	0
–	–	0
–	–	0
–	–	0
–	–	0
–	–	0
–	–	0

Table 6: Example 1 - The Register Map Table p1

Type (3 bits)	*Mem Address (17 bits)*	*Register Map (7 bits)*
sram-wr-xfer	0	b1
sram	128	a0
–	–	0
–	–	0
–	–	0
–	–	0
–	–	0
–	–	0

Table 7: Example 1 - The Register Map Table p2

```
alu[b1, a3, +, b2]
L1: br!=ctx[0, L2]
L2: alu[a0, --, b, b1]
alu[a3, a2, +, b2]
sram[read, $0, 128, 0, 1], ctx_swap
L3: alu[a1, b2, +, $0]
```

The allocator changes the seventh instruction to a move instruction as well. It must also allocate a new GPR for the read transfer register `$0`. The new contents of the CAM table are shown in table 8.

```
L0: alu[a3, a1, +, b2]
alu[b2, a0, +, b0]
alu[b1, a3, +, b2]
L1: br!=ctx[0, L2]
L2: alu[a0, --, b, b1]
alu[a3, a2, +, b2]
alu[b3, --, b, a0]
L3: alu[a1, b2, +, $0]
```

Finally, the allocator maps the use of the read transfer register `$0` in instruction 8 to its GPR located in the CAM table.

```
L0: alu[a3, a1, +, b2]
alu[b2, a0, +, b0]
alu[b1, a3, +, b2]
L1: br!=ctx[0, L2]
L2: alu[a0, --, b, b1]
alu[a3, a2, +, b2]
alu[b3, --, b, a0]
L3: alu[a1, b2, +, b3]
```

Type (3 bits)	*Mem Address (17 bits)*	*Register Map (7 bits)*
sram-wr-xfer	0	b1
sram	128	a0
sram-rd-xfer	0	b3
–	–	0
–	–	0
–	–	0
–	–	0
–	–	0

Table 8: Example 1 - The Register Map Table p5

6 Other IXP Implementation Details

6.1 Transfer Registers

The processor uses transfer registers [4] when transmitting to external devices such as SRAM. Transfer registers are not general-purpose and can only be exclusively read or written. A transfer register must appear as the source of a store operation instruction, but it can not be used as the corresponding source register in the new move instruction. For example, if hardware needs to map address 128 to register A1, transforming

```
sram_write $0, 128, ..., CTX_SWAP
```

into

```
alu_b       A1,  $0
```

the value of $0 in the resulting instruction is the value of the read transfer register $0, not the value of write transfer register.

For the address-register mapping scheme to work, instructions that use transfer registers must replace those registers with general purpose registers. This can mean that 2 general purpose registers are required for each memory-address transfer-register pair, but in practice one transfer register is used for a large number of memory addresses.

Statically, this work needs to account for the case where a transfer register is the source of a memory operation such as `t_fifo_wr`, a write to the transfer FIFO, that is not rewritten by the address-register mapper. The algorithm does this by looking for the next store instruction following a transfer register definition and the previous load instruction following a transfer register use. If the load or store instruction is not a possible remap target, the algorithm notes that in the option field of the corresponding instruction that uses the transfer register. If the transfer register is the source of both types (via a branch), then the program splits the instruction into a mappable and non-mappable version. For example, the following program

```
    immed       $0  1
    sram_write  $0  B0  0  1
    immed       $1  5
    t_fifo_wr   $1  B1  B2 1
    t_fifo_rd   $2  B2  0  1
    alu_add     A1  A3  $2
    immed       $1  7
    br=ctx      0   L0
    t_fifo_wr   $1  B3  0  1
L0: sram_write  $1  B0  0  1
```

transforms into

```
    immed       $0  1
    sram_write  $0  B0  0  1
    immed       $1  5         NO_MAP
    t_fifo_wr   $1  B1  B2 1
    t_fifo_rd   $2  B2  0  1
    alu_add     A1  A3  $2    NO_MAP
    immed       $1  7
    immed       $1  7         NO_MAP
    br=ctx      0   L0
    t_fifo_wr   $1  B3  0
L0: sram_write  $1  B0  0
```

This example shows all of the possible cases. The transfer register instructions that associated with the FIFO operations are annotated with the `NO_MAP` option. The transfer register instruction that is the source of both a `t_fifo_wr` instruction and a `sram_write` instruction is split into two transfer register instructions, one of which has the `NO_MAP` option.

6.2 Unpacking Memory Instructions

A memory instruction can load or store a range of words at once. In that case the transfer register specified in the destination slot only represents the start of the range of transfer registers used in the operation. The hardware needs to unpack the memory operation, so that each transfer register is exposed in the instruction stream.

For example,

```
    sram_write  $0  B0  0  4
```

transforms into

```
    sram_write  $0  B0  0  1
    sram_write  $1  B0  1  1
    sram_write  $2  B0  2  1
    sram_write  $3  B0  3  1
```

Benchmark	*# SRAM Loads Pre-Opt*	*# SRAM Loads Post-Opt*	*% Decrease*
ipfdwr (1 ME)	339643	301484	11%
ipfdwr (4 ME)	1284455	1134751	12%
md4 (1 ME)	674500	571901	15%
md4 (4 ME)	1739127	1471492	15%
nat (1 ME)	491110	364786	25%
nat (4 ME)	732668	568279	23%
url (1 ME)	860114	785074	9%
url (4 ME)	2612944	2407754	8%

Table 9: Dynamic Load Count

Benchmark	*# SRAM Stores Pre-Opt*	*# SRAM Stores Post-Opt*	*% Decrease*
ipfdwr (1 ME)	91515	74127	19%
ipfdwr (4 ME)	351549	283618	19%
md4 (1 ME)	320169	256253	20%
md4 (4 ME)	894918	187932	21%
nat (1 ME)	207839	149644	28%
nat (4 ME)	346453	261299	25%
url (1 ME)	132049	94208	12%
url (4 ME)	420058	365450	13%

Table 10: Dynamic Store Count

7 Results

This paper uses a subset of the Netbench [2] benchmark suite for its results. We could only use a small subset of the suite that has been ported to NePSim [10].

Each benchmark executes forever, the first 8000000 instructions run on each benchmark before halting. The benchmarks are run on all 6 microengines (4 intermediate MEs) and on 3 microengines (1 intermediate ME).

Tables 9 and 10 show the SRAM dynamic load+store counts before and after optimization. There are two factors which limit the number of load+stores which can be removed: 1) Most importantly, the optimization requires many registers, while the number of available dead registers is limited. 2) Some of the load+store activity facilitates inter-engine communication; these cannot be removed. The store numbers are universally better than the load numbers because any store can be immediately allocated to a register, while a load requires that a corresponding store is already allocated to a register.

Table 11 shows the reduction in idle cycles. The idle cycle reduction varies between benchmarks. However, we observe a correlation between the dynamic

Benchmark	*Pre-Opt*	*Post-Opt 8-cycle lat*	*Decrease*	*Post-Opt 10-cycle lat*	*Decrease*
ipfdwr (1 ME)	61	139	-221%	334	-548%
ipfdwr (4 ME)	88217	–	–	49019	55.6%
md4 (1 ME)	629387	469737	25.4%	–	–
md4 (4 ME)	9574894	9470987	1.1%	9565548	0.1%
nat (1 ME)	101284	26676	73.7%	50443	50.2%
nat (4 ME)	106866	38325	64.1%	73344	31.4%
url (1 ME)	1006534	860238	14.5%	1058777	-5.2%
url (4 ME)	7728240	7079193	8.4%	7376432	4.6%

Table 11: Idle Cycle Count

load+store count and the idle cycle reduction. Also, the relative number of idle cycles removed decreases when the number of microengines increases. This is probably due to the increase in cross-microengine communication.

We also performed experiments that change the amount of latency associated with the new hardware. If, for instance, the hash table lookup and the CAM table lookup can not be done in parallel, the total latency increases from 8 cycles worst-case to 10 cycles worst-case (I assumed that the hashing and retrieval can be completed in 2 cycles). The idle cycle reduction for 10-cycle worst-case latency hardware is obviously worse than the idle cycle reduction for 8-cycle worst-case latency hardware, but overall the performance gain is still good.

There is a 50% reduction in idle cycles for the nat benchmark.

Table 12 shows the increase the packet throughput for each benchmark. Intuitively a decrease in idle cycles should cause an increase in throughput performance. The throughput however is a complicated function of many parameters not just idle cycles. We examined the benchmarks and the throughput is mainly a function of the design and inter PE communication which is not handled by our framework. In some cases, the idle cycles form a part of the critical path and our framework optimized it away significantly. For example, there is a 14% increase in the speed of the nat benchmark, which is promising.

Table 13 shows the different reasons that the algorithm is unable to remove all spills. Cross-communication indicates a memory access that is inherently unremovable with the current IXP hardware. It is a memory access the programmer uses for communication with another microengine rather than for storage purposes. The more important reason that the hardware cannot remove a spill is due to size restrictions. The program only has a limited amount of dead registers, furthermore the CAM table can only hold 8 different addresses simultaneously.

Benchmark	*#Pkts Pre-Opt*	*#Pkts Post-Opt*	*% Increase*
ipfdwr (1 ME)	18297	18701	2.21%
ipfdwr (4 ME)	70302	75490	7.38%
md4 (1 ME)	4210	4485	6.53%
nat (1 ME)	28645	32755	14.35%
nat (4 ME)	44898	50101	11.59%
url (1 ME)	2174	2245	3.27%
url (4 ME)	7139	7387	3.47%

Table 12: Throughput

Benchmark	Removed Spills	Cross-Communication	Out of Space
ipfwdr (1 ME)	55547	22248	353363
ipfwdr (4 ME)	217635	31599	1386770
md4 (1 ME)	166515	52399	775755
md4 (4 ME)	974621	181459	1477965
nat (1 ME)	184519	33410	481020
nat (4 ME)	249543	81628	747950
url (1 ME)	112881	47425	831857
url (4 ME)	259798	204846	3292800

Table 13: The different types of memory accesses

8 Related Works

[11] is the work most related to this paper. That paper introduces the concept of splitting the register file into shared and private registers. A shared register must be dead across all context switch points. This paper extends [11] by the relaxing the constraint that the shared register must always be dead across context switch points.

[3] presents an alternative scheme for IXP register allocation. It uses Integer Linear Programming to solve to optimally allocate the registers based on the constraints set for each register type. This static technique performs well in practice, but they do not address the issue of threads in their paper or inter-thread allocation.

Register renaming is an old concept in superscalar processors [8] [7]. There are two key differences between the renaming mechanism presented here and the renaming unit in a superscalar processor. The hardware presented here attempts to rename memory addresses to register, rather than renaming virtual registers to physical registers. Also, the goal here is to reduce memory activity, while traditionally the goal is to remove data dependencies between instructions, increasing the amount of parallelism.

9 Conclusion

In conclusion, the dynamic register allocation approach presented here attempts to go beyond the best statically available allocation techniques, by combining static analysis with dynamic allocation. By dynamically mapping memory addresses onto registers, it can reduce the total number of dynamic memory operations. In turn, reducing the total number of memory operation reduces the idle cycle count, which is the goal of all optimizations for systems with real-time constraints.

The results show that this approach is able to reduce idle cycle counts in all benchmarks and achieve an unweighted average decrease of 51% in idle cycles with a 8 cycle latency. These results also show that idle counts can be reduced even further if hardware supporting Next Neighbor registers is available. The hardware overhead introduced by our method is insignificant and is off the critical path. This paper shows that it is viable to use smart dynamic allocation techniques over existing static algorithms.

Our current work is focused on getting around the limitations of running out of dead registers. In particular, we are developing dynamic deadness detection mechanisms to assist in this regard by combining static analysis with dynamic information.

References

[1] Qi Fan Chen Edward A. Fox, Lenwood S. Heath and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *CACM*, Jan-

uary 1992.

[2] W. Hu. G. Memik, W.H. Mangione-Smith. Netbench: A benchmarking suite for network processors. *In Proceeding of the International Conference on Computer Aided Degin*, June 2001.

[3] Lal George and Matthias Blume. Taming the ixp network processor. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[4] Intel Corporation. *Intel IXP1200 Processor Family - Reference Manual*, December 2001.

[5] Intel Corporation. *Intel IXP2800 Processor Family - Reference Manual*, August 2004.

[6] Y. Park J. Kim, S. Jung. Experience with a retargetable compiler for for a commericial network processor. *In Proceeding of Internation Conference on Compilers, Architecture and Synthesis for Embedded Systems*, October 2002.

[7] Mateo Valero et. al. Teresa Monreal, Antonio González. Delaying physical register allocation through virtual-physical registers. *In Proceedings of the 32nd Annual International Symposium on Microarchitecture*, 1999.

[8] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, January 1967.

[9] J. Wagner and R. Leupers. C compiler design for an industrial network processor. *In Proceedings of ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, June 2001.

[10] Laxmi Bhuyan Yan Luo, Jun Yang and Li Zhao. Nepsim: A network processor simulator with power evaluation framework. *IEEE Micro Special Issue on Network Processors for Future High-End Systems and Applications*, 2004.

[11] Xiaotang Zhuang and Santosh Pande. Balanced register allocation across threads for a multi-threaded network processor. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.