Compiler Assisted Dynamic Management of Registers for Network Processors

Ryan Collins[1], Fernando Alegre[1], Xiaotong Zhuang[1], and Santosh Pande[1]

[1]Georgia Institute of Technology
College of Computing
Atltanta, GA 30332-0280 USA
{rcollins, fernando, xt2000, santosh}@cc.gatech.edu

Abstract

Modern network processors support high levels of parallelism in packet processing by supporting multiple threads that execute on a micro-engine. Threads switch context upon encountering long latency memory accesses and this way the parallelism and memory access can be overlapped. Context switches in the typical network processor architectures such as the IXP are designed to be very fast. However, the low overhead is partly achieved by leaving register management to programs, with minimal 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 a compiler. However, a purely static analysis is unable 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 the transfer registers and addresses to dead general purpose registers and update the usage of 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 on Intel IXP 1200 network processor.

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. In this way, parallelism is achieved by interleaving the execution of a thread with memory access for the others. 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 using high level languages [12, 6, 1, 2]. Typical network processors have 128 registers available which can used in a shared or non-shared manner across the threads. A 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. [13] 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 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.

This paper is organized as follows. In section 2, we briefly discuss the architecture of the processor we base our work upon and discuss the semantics of key instructions. In section 3, we provide the motivation for our approach, which is described in sections 4,5 and 6. Section 7 shows our results, and finally sections 8 and 9 provide references to related works and discussion about our results, respectively.

2 The Intel IXP1200

In this paper, we used the Intel IXP1200 processor as the target [1]. This processor consists of multiple RISC cores (*microengines*) which may work either in parallel or in a pipeline. Microengines share a common bus with a number of components.

2.1 Key Components

The key components relevant to this paper are explained below. There are several additional components present in the IXP1200 which we omitted because they are not used in this work.

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

Microengines: Microengines are processors with a small instruction set and hardware support for cooperative (not pre-emptive) multithreading. Each microengine has a small program code store shared by all threads and a separate program counter for each thread. The IXP1200 supports 4 threads per microengine.

ALU and GPRs: Each microengine has an arithmetic and logic unit (ALU) and a set of general-purpose registers, which are evenly divided into 2 banks (known as A and B.) Upon a context switch, contents of registers are not saved. Therefore, context-switches are very fast, but register file management must be done by the software.

Transfer Registers: Each microengine has special purpose registers to communicate with memory units, one such set for each type of memory (e.g., SRAM or SDRAM) and transfer direction (read/write.) These are the only means to move data between the ALU and memory.

Memory Units: Memory units are shared by all microengines. At least 2 different types of memory are present: SRAM, with a typical latency of 20 cycles or more, and SDRAM, with a latency of 50 cycles or more.

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. Thus, the effect of memory latency is diminished.

2.2 IXP Assembly Instructions

Most IXP-architecture assembly instructions follow the format:

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

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

Arithmetic and Logic Unit: Arithmetic, logic and register-to-register copy operations use the keyword `alu`. The first argument typically indicates the destination operand, which may be either a general-purpose register or a transfer register. It is followed by the operator keyword [2] and the source operand. Not all combinations are legal. In particular, general-purpose registers must be in different banks.

General-purpose registers: General-purpose registers in each bank can be addressed either absolutely (prefixed with `@`) or relative to each thread (no prefix.) In the IXP1200, each thread can address 16 registers per bank. For example, 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.

Transfer registers: 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.

Memory: Programmers are required to determine where to store (e.g., `sdram` or `sram`) a particular piece of data, since there is nothing equivalent to the transparent cache hierarchy found in modern general-purpose computers.

Since it is not possible to move data directly between a general-purpose register and memory, typical data transfers take place through transfer registers as follows:

```
sdram[read,$$1, b2, 0, 1]   // SDRAM[b2] to sdram-xfer
alu[a0,--,B,$$1]            // Copy data into GPR A0
...
alu[$0,--,B,b5]             // GPR to sram-xfer
sram[write, $0, a1, 0, 1]   // Write to SRAM[a1]
```

Control flow: Other instructions used below are branching (`br`) and context switches (`ctx_arb`,) which may be either unconditional or conditional.

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 enters 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.

[2]In copy operations, the operator field is empty

Benchmark	*Number*	*% 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. Idle Cycles

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

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.

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

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

Taking as input a binary file, we perform static analysis to produce an annotated version of the file. We assume 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, our algorithm follows [13] to create a control flow graph (CFG) that divides the blocks into non-switch regions. These are basic blocks that have been sub-partitioned to include at most one context-switch instruction at its boundaries, just as normal basic blocks contain at most one branch instruction (cf. figure 1.)

Next, data analysis is performed to discover dead-until-end registers, i.e., registers that are dead for the duration of the program. The following dataflow equations are used:

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

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

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

Table 3. Memory Access Patterns

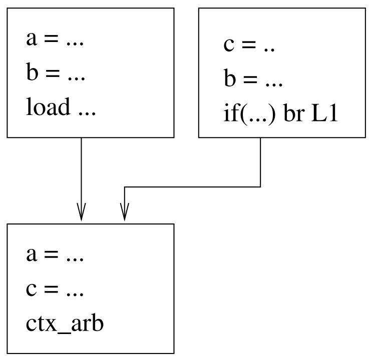

Figure 1. A non-switch region CFG

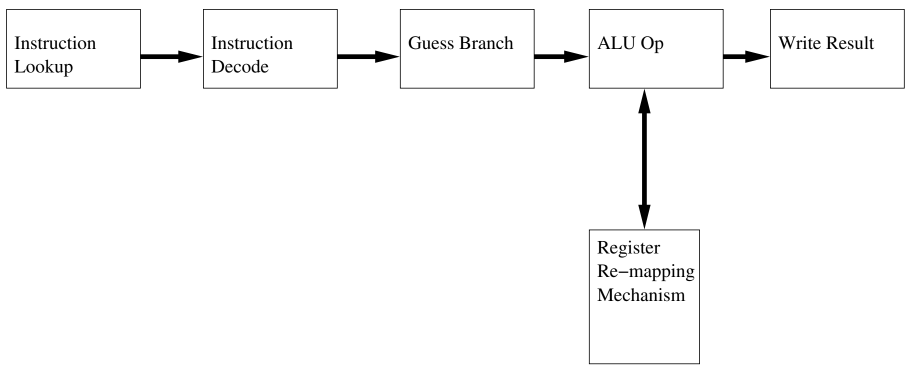

Figure 2. The modified IXP pipeline

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, in this code,

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

traditional liveness analysis would consider `a14` dead after its use in the second line. However, when multiple threads are present, it is unsafe to treat `a14` as dead for the rest of the program. Furthermore, 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 which would benefit the most from our algorithm.

The dead/not-dead status of all registers is encoded as a bit vector at several points in the program. These bit vectors are collected into a hash-table, indexed by the program point, which will be loaded into SRAM at program initialization time. The length of the vector can be either 128 bits if absolute addressing is used in the program, or 32 bits if only relative addressing is used.

Finally, our algorithm needs to analyze the locality of memory operations. We need to distinguish operations used for spill values from those used for communication between microengines. Our dynamic allocator cannot handle the latter due to inefficiencies of the IXP1200 model, which lacks more advanced communication mechanisms such as the Next Neighbor registers found in the Intel IXP2800 processor. Therefore, each memory operation is statically annotated with a bit indicating whether it is used globally or locally.

`alu[$0,a3,+,b4]`
`sram[write,$0,128,0,1]`
`ctx_arb`
`...`
`sram[read,$0,128,0,1]`
`ctx_arb`
`alu[a1,b2,+,$0]`
	`alu[a16,a3,+,b4]`
	`alu[a15,--,b,a16]`
	`...`
	`alu[a17,--,b,a15]`
	`alu[a1,b2,+,a17]`
		`alu[a15,a3,+,b4]`
		`alu[$0,--,b,a15]`
		`sram[write,$0,128,0,1]`
		`ctx_arb`
		`...`
		`sram[read,$0,128,0,1]`
		`ctx_arb`
		`alu[a1,b2,+,$0]`

Table 4. Dynamic code transformation. Left: Original code. Middle: When A15-A17 are dead. Right: When only A15 is dead.

5 Dynamic Implementation

Let us now show how a small modification to the hardware could use the register usage information from the previous section to dynamically map memory addresses to dead registers.

Table 4 shows an example of such transformation both when there are enough dead registers available and when there are not. In the latter case, our hardware solution would greedily allocate the dead register to the next address that required it (the write transfer register `$0` in the example.) This transformation would preserve 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. (cf. Figure 2). 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 3 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

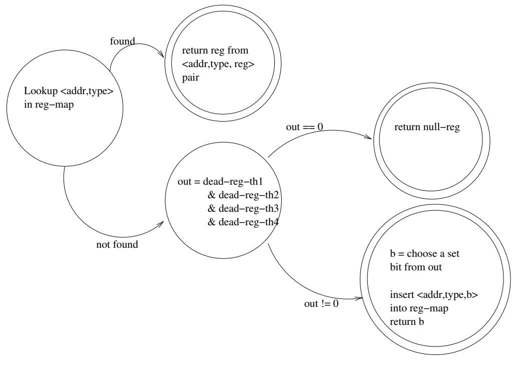
Lookup <addr,type> in reg−map
found
return reg from <addr,type, reg> pair
not found
out = dead−reg−th1 & dead−reg−th2 & dead−reg−th3 & dead−reg−th4
out == 0
return null−reg
out != 0
b = choose a set bit from out
insert <addr,type,b> into reg−map return b

Figure 3. Dynamic Allocation FSM

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

Table 5. Sample CAM entry

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

Table 6. The 64-entry hash table

```
{1} alu[a3,a1,+,b2]
{2} alu[b2,a0,+,b0]
{3} alu[$0,a3,+,b2]
{4} br!=ctx[0,L1]
{5} L1:sram[write,$0,128,0,1]
    ctx_arb
{6} alu[a3,a2,+,b2]
{7} sram[read,$0,128,0,1]
    ctx_arb
{8} alu[a1,b2,+,$0]
```

Original code

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

Step 1

```
alu[a3,a1,+,b2]
alu[b2,a0,+,b0]
alu[b1,a3,+,b2]
br!=ctx[0,L1]
L1:alu[a0,--,b,b1]
alu[a3,a2,+,b2]
sram[read,$0,128,0,1]
ctx_arb
alu[a1,b2,+,$0]
```

Step 2

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

Step 3

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

Final version

Table 7. Example I. Dynamic code transformation

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.

A new store consisting of a CAM table and a hash table is needed. The 8-entry CAM table is a mapping from a memory address to a register containing its cached value. The 64-entry hash table is a mapping from a program point (PC) to a bit vector marking the dead registers available at that program point.

Each CAM entry contains the type of memory acccessed along with the address. The type tag also doubles as a valid bit, with type 0 corresponding to the invalid state. The memory is then bypassed by accessing the register in the Register Map entry. In the case of table 5, register `A3` points to address 0xF000, which is cached in register `B4`.

If the current address is not in the lookup table, the hardware allocates an unused register. It determines that a register is a candidate if it appears in the intersection of the dead-rest bit-vectors for all four threads at the most recent context switch. The harware then chooses among those the register corresponding to the lowest order bit in the vectors.

Table 6 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 [3] 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 keep a 128-bit dead-register bit-vector.

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 this solution is off the critical path and 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 is 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, a context switch is avoided. When not found, the latency overhead added would still be only 8 cycles.

We now present a full example to illustrate the technique.

Example I. Suppose 4 registers in the A bank and 4 registers in the B bank are available, and we are given the code shown in table 7.

By static analysis, we see that registers b1 and b3 are dead before non-switch region `{1-5}`, a0, b0, b1 and b3 are dead before `{6-7}` and a0,a2,a3,b0,b1 and b3 are dead before `{8}`.

When the hardware reaches `{3}`, it must replace the register `$0` with a general purpose register (cf. Step 1 in table 7 and CAM entries in table 8.) Then, the dynamic register allocator changes `{5}` to a new move instruction (cf. Step 2 in table 7.) Then, the allocator changes `{7}` to a move instruction as well. It must also allocate a new GPR for the read transfer register `$0` (cf. Step 3 in table 7, table 9.) Finally, the allocator maps the use of the read transfer register `$0` in instruction `{8}` to its GPR located in the CAM table. The final transformed code is also shown in table 7

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

Table 8. Example I. Register Map Table p1

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

Table 9. Register Map Table p5

```
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]
```

```
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,b3,0,1]
L0: sram[write,$1,b0,0,1]
```

Table 10. Example II. Left: Original code. Right: Transformed code

6 Other IXP Implementation Details

6.1 Transfer Registers

As we mentioned above, the processor uses transfer registers when transmitting to external devices such as SRAM. Transfer registers, which cannot be used for general purposes, are either read-only or write-only. Thus, we also need to cache the values stored in write-only registers, so that we avoid going through the process of writing them to memory and back to a read-only register. In summary, for the address-register mapping scheme to work, instructions that use transfer registers must replace those registers with general purpose registers. This may mean that 2 general purpose registers are required for each memory-address transfer-register pair. In practice, however, one transfer register is used for a large number of memory addresses.

During static analysis, we also need to account for the case where a transfer register is the source of other memory operations, such as `t_fifo_wr` (a write to the transfer FIFO,) that are not rewritten by the address-register mapper. Our 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. An example is given in table 10.

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 represents only 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]
```

7 Results

This paper uses a subset of the Netbench [8] benchmark suite for its results. We could only use a small subset of the suite that has been ported to NePSim [7], the IXP1200 simulator in which we tested our work.

Each benchmark consists of an infinite loop. We had each benchmark execute for 8000000 instructions before halting. The same benchmarks were run in two configurations: all 6 microengines (4 intermediate MEs) and 3 microengines (1 intermediate ME).

Table 11 shows the SRAM dynamic load+store counts before and after optimization. There are two factors that limit the number of load+stores that 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, and these cannot be removed. The store numbers are universally better than the load numbers because any store can

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 13. Throughput

be immediately allocated to a register, while a load requires that a corresponding store be already allocated to a register.

Table 12 shows the reduction in idle cycles. The idle cycle reduction varies between benchmarks. However, we observe a correlation between the dynamic 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 (assuming 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 13 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 14 shows the different reasons that the algorithm is unable to remove all spills. Cross-communication indicates a memory access that is in-

Benchmark	*Dynamic Load Count*			*Dynamic Store Count*		
	Pre-Opt	*Post-Opt*	*Decrease*	*Pre-Opt*	*Post-Opt*	*Decrease*
ipfdwr (1 ME)	339643	301484	11%	91515	74127	19%
ipfdwr (4 ME)	1284455	1134751	12%	351549	283618	19%
md4 (1 ME)	674500	571901	15%	320169	256253	20%
md4 (4 ME)	1739127	1471492	15%	894918	187932	21%
nat (1 ME)	491110	364786	25%	207839	149644	28%
nat (4 ME)	732668	568279	23%	346453	261299	25%
url (1 ME)	860114	785074	9%	132049	94208	12%
url (4 ME)	2612944	2407754	8%	420058	365450	13%

Table 11. SRAM Counts

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 12. Idle Cycle Count

herently 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.

8 Related Works

[13] 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 [13] by the relaxing the constraint that the shared register must always be dead across context switch points.

[4] 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 [11] [9]. 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.

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 14. Occurrences of memory accesses by type

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] M. K. Chen, X. F. Li, R. Lian, J. H. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: achieving high performance from compiled network applicati ons while enabling ease of programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 224–236, New York, NY, USA, 2005. ACM Press.

[2] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipeline d architectures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 237–248, New York, NY, USA, 2005. ACM Press.

[3] E. A. Fox, L. S. Heath, Q. F. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. *Commun. ACM*, 35(1):105–121, 1992.

[4] L. George and M. Blume. Taming the ixp network processor. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press.

[5] J. Hasan, S. Chandra, and T. N. Vijaykumar. Efficient use of memory bandwidth to improve network processor throughput. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 300–313, New York, NY, USA, 2003. ACM Press.

[6] J. Kim, S. Jung, Y. Paek, and G.-R. Uh. Experience with a retargetable compiler for a commercial network processor. In *CASES '02: Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 178–187, New York, NY, USA, 2002. ACM Press.

[7] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao. Nepsim: A network processor simulator with a power evaluation framework. *IEEE Micro*, 24(5):34–44, 2004.

[8] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: a benchmarking suite for network processors. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 39–42, Piscataway, NJ, USA, 2001. IEEE Press.

[9] T. Monreal, A. Gonzalez, and M. V. et al. Delaying physical register allocation through virtual-physical registers. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 186–192, Washington, DC, USA, 1999. IEEE Computer Society.

[10] T. Sherwood, G. Varghese, and B. Calder. A pipelined memory architecture for high throughput network processors. In *ISCA '03: Proceedings of the 30th annual international symposium o n Computer architecture*, pages 288–299, New York, NY, USA, 2003. ACM Press.

[11] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *Instruction-level parallel processors*, pages 13–21, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[12] J. Wagner and R. Leupers. C compiler design for an industrial network processor. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 155–164, New York, NY, USA, 2001. ACM Press.

[13] X. Zhuang and S. Pande. Balancing register allocation across threads for a multithreaded network processor. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 289–300, 2004.