Balancing Register Allocation Across Threads for a Multithreaded Network Processor

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

ABSTRACT

Modern network processors employ multi-threading to allow concurrency amongst multiple packet processing tasks. We studied the properties of applications running on the network processors and observed that their imbalanced register requirements across different threads at different program points could lead to poor performance. Many times application needs demand some threads to be more performance critical than others and thus by controlling the register allocation across threads one could impact the performance of the threads and get the desired performance properties for concurrent threads. This prompts our work.

Our register allocator aims to distribute available registers to different threads according to their needs. The compiler analyzes the register needs of each thread both at the point of a context switch as well as internally. Compiler then designates some registers as shared and some as private to each thread. Shared registers are allocated across all threads explicitly by the compiler. Values that are live across a context switch can not be kept in shared registers due to safety reasons; thus, only those live ranges that are internal to the context switch can be safely allocated to shared registers. Spill can cause a context switch. and thus, the problems of context switch and allocation are closely coupled and we propose a solution to this problem. The proposed interference graphs (GIG,BIG,IIG) distinguish variables that must use a thread's private registers from those that can use shared registers. We first estimate the register requirement bounds, then reduce from the upper bound gradually to achieve a good register balance among threads. To reduce the register needs, move insertions are inserted at program points that split the live ranges or the nodes on the interference graph. We show that the lower bound is reachable via live range splitting and is adequate for our benchmark programs for simultaneously assigning them on different threads. As our objective, the number of move instructions is minimized.

Empirical results show that the compiler is able to effectively control the register allocation across threads by maximizing the number of shared registers. Speed-up for performance critical threads ranges from 18 to 24% whereas degradation for performance of non-critical threads ranges only from 1 to 4%.

PLDI'04, June 9–11, 2004, Washington, DC, USA.

Categories and Subject Descriptors:

D.3.4 [Programming Languages]: Processors—Optimization, Code generation, Run-time environments.

General Terms: Algorithms, Languages, Performance.

Keywords: Network Processor, Register Allocation, Multithreaded Processor.

1. INTRODUCTION

The dramatic growth in Internet traffic has motivated a specialized category of embedded processors called *Network Processors (NPs)* with fast processing speed and specialized hardware support for network applications. Network processors are distinguished by their fast processing core and are programmed in a dedicated manner for catering to the specific needs of underlying applications. The compiler optimization for network processor is an emerging topic for research [3][4][5][19]. In this paper, we attempt the register allocation problem for a multithreaded network processor IXP. The IXP's network processor model can be applied to any network processor with shared CPU and register file for multiple threads and with fast context switch to hide long latency operations such as memory accesses. Typically, network processor applications consist of multiple threads concurrently executing multiple tasks of a network processing application. The tasks can be as simple as packet routing to complex ones that process packet contents for viruses and malignant code etc. In contrast to general processors, the tasks that execute on different threads of a network processor are bound to them at compilation time; in other words, no run time thread assignment takes place. Since low level operations originally done with OS or hardware such as context switch are exposed to the programmer, the compiler has the knowledge of thread interactions which are predictable. It is obvious that different tasks have different complexities and also levels of desired performance. Some tasks may be more performance critical than others. Implementing (effecting) such performance needs across threads is currently impossible for any user. This is so since compiler allocates a fixed number (32) of registers to each thread and does not undertake inter-thread analysis to balance their overall register needs. It may be noted that performance of a thread is quite sensitive to register needs; even though the number of spills may be small for a larger number of registers, each spill is very expensive (latency of about 20 cycles). Our experience with Intel's IXP network process family, which largely follows this model, tells us that: 1) we can achieve register balancing among different threads and 2) we can reduce spills through the safe use of shared registers which are not

live across context switch instructions for individual thread 3) through the use of register sharing, overall, we make more registers available to threads boosting their performance. Thus, overall by balancing register needs across threads we can meet their performance requirements. These optimizations are necessary due to the disparity of register pressures across threads and across different regions of code in each thread. We first discuss the network processor architectures to gain some understanding of the problem of balancing register requirements across threads.

1.1 Network processors

State-of-the-art network processors like Intel IXP1200/2400/2800, MMC nP series, IBM power NP etc.[21]. have programmable processing core that can be coded for application needs. In contrast to traditional processors, network processors have their special properties.

Speed vs Flexibility

Network processors face the dilemma of offering both prompt processing of the network traffic and flexibility to the software programmers to meet the requirements of different applications.

As the network speed continues to increase, the time to process each packet must be shortened to avoid packet loss. For example, processing at OC-192 allows only 52ns for each packet and OC-768 leaves only 13ns for processing. The higher speed requires both shorter processing time for each packet and shorter time a packet can stay in the system (waiting time + processing time).

To speedup the critical paths for packet processing, normally a number of RISC processor cores are equipped to work in parallel. Although sometimes a co-processor (typically a general purpose processor) is added to handle other slow tasks, the packet processing core must be optimized for speed. Therefore, features such as explicit multithreading, explicit and fast context switching (only pc is saved), direct memory access (without the complication of caches) are commonly seen in network processor designs. As memory operations are extremely time-consuming, solutions should focus on hiding the latency with tolerable hardware and software complexity. With fast context switch, each processor core can hide latencies by context switching to other threads when accessing the peripherals. Even if caches are enabled, context switch to other threads is generally a clever way to avoid the deviation in memory access time like in the MMC nP series.

Network processors are also aimed to provide plethora of solutions for network applications, which were originally implemented with dedicated hardware (not flexible) or general purpose processors (too slow). Recent research [1][2][22] has attempted complicated tasks such as content inspection, software routers, intrusion detection, etc. As more code base is added to network processors, writing in assembly can be error-prone and time-assuming, which greatly hampers the fast prototyping and increases time to market. To provide programmability, a High Level Language (HLL) compiler is sometimes provided, although typically with limited language feature support. There are several on-going research efforts to build proper optimizing compiler for network processors [3][4][5]. As mentioned earlier, non of the current compilers undertake inter-thread analysis forcing programmers to manage the register pressure across threads. Without any help from the compiler it is impossible for a user to hand-tune (multi-threaded) code. This mot

Intel IXP Network Processor

In this paper, we base our work on the Intel IXP network processor. Since its successful design has made it a very popular product in the network processor market, we generalize its features as a general model in section 2. Here, we present several prominent features of the IXP network processor, which prompt the thread register allocation problem (details in section 1.2).

Figure 1 shows the block diagram of the IXP1200 network processor. The chip has 6 micro-engines (*processing units or PU*) and 4 threads share the same PU. The chip has connections to off-chip SRAM, SDRAM, PCI bus etc. As shown in Figure 2.a, typically, each PU gets packets from its input queues, processes it and then writes to its output queues, or the input queues of the next PU in the next pipeline stage. With pipeline processing, typically, some PUs are in charge of getting packets from the input ports; some handle packet processing and some are for output ports.

Our optimization focuses on the code on different threads of the same PU. Figure 2.b shows major components inside each PU. Some of the important features are as follows:

1. *Shared register file but typically non-overlapped partitions*. Figure 2.b shows that the general purpose register (GPR) file is shared by the 4 threads. Each thread has access to all registers; however without optimization, each thread is normally allocated non-overlapping part of the register file. The reason for the register file partition is due to light-weighted context switch as discussed below.
2. *Non-preemptable thread execution*. There is no operating system, no control present over the threads sharing the CPU. A thread gives up the CPU only when it blocks on I/O or other long latency operation or executes a context switch (ctx_switch) instruction voluntarily[1].
3. *Light-weighted context switch*. Context switch is cheap (only PC is saved), this is also the reason registers are normally allocated in a non-overlapped fashion from the register file. If a register is allocated to two threads, after context switch, the content in that register may be modified by the other thread. Since registers are neither automatically saved nor restored during a context switch such possibilities exist and this is where it becomes a compiler problem to manage registers.
4. *Cheap ALU, expensive memory access*. No cache is available for memory accesses; at least 20 cycles are needed for each load/store instruction. Context switches are typically followed to hide the long latency of memory accesses. In contrast, all ALU instructions can be completed in 1 cycle. Large memory latency makes overall performance sensitive to spills even though they may be few in number.

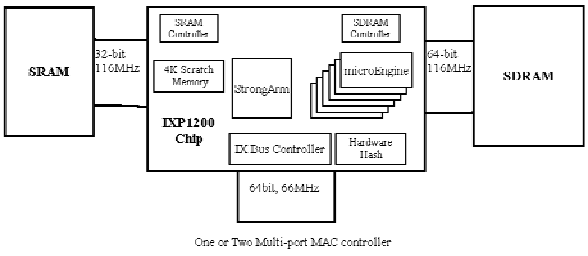

Figure 1. IXP1200 block diagram.

[1] ctx_switch instruction can be inserted by the programmer to achieve fair sharing of the CPU.

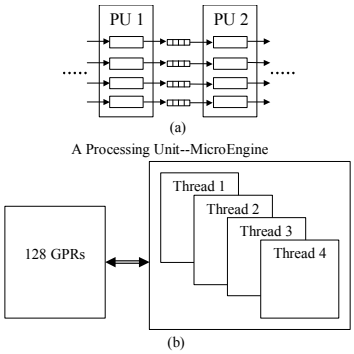

Figure 2. IXP1200 threads and register file on a PU.

The above features of the IXP network processor are driven by design philosophy to simplify hardware so as to increase the clock rate and execution speed. For instance, context switch is kept very simple and fast (1 cycle latency). For this only program counter (pc) is saved but no registers are saved because it can cause long delay in context switch which may offset the benefits of CPU sharing. On the other hand, since all the hardware details are exposed, compiler can prudent decisions regarding register sharing etc. Next, we propose the multi-threaded register allocation problem.

1.2 The Register Allocation Problem

As mentioned above, although the register file can be accessed by all threads, it has to be partitioned without overlap across threads because no register is saved/restored during context switch. Here, we argue that some registers can be *safely* shared by all threads through compiler analysis since thread switch is predictable.

The example in Figure 3 illustrates the problem and the possible ways to solve it. In Figure 3.a, the code for two threads are shown. Assume all variables are dead after their last use in the code. In thread 1, a code segment contains 12 instructions, including two context switch instructions—ctx_switch gives up CPU voluntarily and a load causes context switch to wait for I/O operation. Any pair of the 3 variables interferes with each other (co-live at some program points), so in Figure 3.b, they are assigned 3 different physical registers. Notice that variable a is live across ctx_switch instruction, so it must be allocated to a physical register that is not used by any other thread, because when thread 1 is context switched at this point, other thread should not modify the physical register of variable a, which means only thread 1 should use the register. On the contrary, variable b and c are only used between two context switch instructions. In other word, when thread 1 is switched out of the CPU, both b and c must be dead. Therefore it is safe to reuse the physical registers allocated to b and c in other threads. Thread 2 has 4 instructions, with two context switch instructions. d is only live between two context switch instructions, therefore d can share a physical registers with other threads. Simply, r2 is shared – used for b in thread 1 and d in thread 2, because the code guarantees that when context is switched to thread 2, r2 contains a dead value (b) for thread 1. Similarly, when context is switched to thread 1, r2 contains a dead value (d) in thread 2. This example shows benefits of sharing registers and lowering total register requirements from four to three. We now show that through another technique (live range splitting) one can reduce total register requirement further.

Three registers seem necessary for thread 1, however we notice that at any program point, only two variables are co-live. This prompts our technique of splitting one of the variables and inserting a move instruction at certain point. This is demonstrated in Figure 3.c. In instruction 6, r3 is replaced by r1, while from instruction 8 to 9, r3 is replaced by r2. Instruction 10 copies r2 to r1, so in instruction 12, we have a consistent replacement (r3→r1). We have managed to reduce total register requirements down to two now.

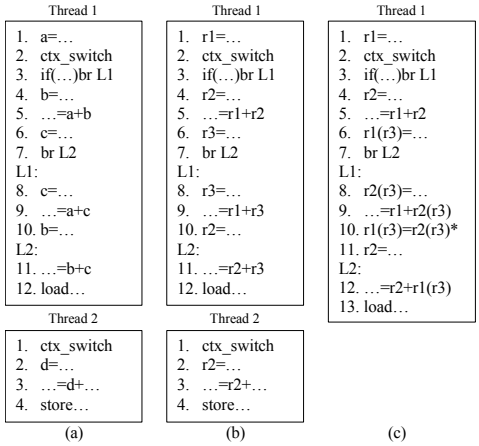

Figure 3. Example of register sharing and move insertion.

The above example illustrates the potential benefits of register sharing across threads and live range splitting. To further justify the multi-threaded register allocation is important and a compiler solution is feasible, we list some properties of the programs that run on the networks to support this argument.

1. For IXP1200, the hardware provides seemingly enough registers. 128 general purpose registers (GPRs) can be used for each PU. However, for each thread, only 32 GPRs are available if no GPR is shared across threads. Register sharing in IXP is a purely software solution, unlike some SMTs (Simultaneous Multi-threading) where it is hardware managed. Compiler designates and allocates a register either as a shared or private one.
2. Since there is no operating system to manage threads, memory access, context switch etc. are all explicit and thus context switch is predictable at compile time.
3. As shown in our experiments, context switch instructions are typically less than 10% of the total instructions and many variables are not live across context switch instructions.
4. PUs are assigned with different tasks. Packets are processed in pipeline fashion--Figure 2.a. Currently, task assignment cannot be done automatically. Although in most cases, the same task is assigned to threads on the same microengine. This actually leads to low utilization of the CPU, because it is hard to chop tasks properly so that they all take roughly ¼ of the computation power of the PU. Therefore, we should assume tasks might be different for threads on the same PU.

Item 1 indicates that the registers may not be sufficient on the network processor. Item 2 and 3 support the feasibility of a compiler solution to optimize the register allocation. Finally, item 4 prompts two kinds of problems, i.e. symmetric vs. asymmetric register allocation, which will be defined in next section.

This paper is organized as follows. Section 2 describes the system model and problem formation, section 3 talks about the construction of the interference graphs, section 4 is the overall framework, section 5 proposes the algorithm to estimate bounds of

register numbers, section 6 and 7 are for inter-thread and intra-thread register allocation, section 8 mentions SRA problem briefly, section 9 shows performance evaluation results and section 10 talks about related work and section 11 is the conclusion.

2. PRELIMINARIES

System Model

In this paper, we study a multithreaded network processor that can run multiple threads on a single processing unit (PU i.e. micro-engine for IXP). The threads on one PU share the computation power of the PU and register files etc. Formally, the model is as follows:

1. There are totally N_{reg} registers that can be used by N_{thd} threads sharing a single PU.
2. Explicit context switch. A thread won't give up the CPU once it starts execution on it, until a context switch instruction is met. Context switch can happen due to explicit instruction or long latency instructions like a load or a store.
3. Context switch is very cheap (only pc is saved) and it is intended to hide long latency operations.
4. Since network packets are mostly independent of each other so are threads. The purpose of multithreading on the same PU is mainly for latency hiding and concurrency. When one thread is stalled due to I/O or other long latency operations, other thread can take the CPU. Therefore, code on different threads are almost independent (Figure 2.a). Thread communication or synchronization rarely happens, however, our current solutions still works under such circumstances. As a future work, knowledge about thread communication or synchronization might be exploited to improve the register allocator.
5. All registers are accessible by all threads, but the registers used by one thread at the point of context switch should not be used anywhere by other threads (later, we will define these registers as *private registers*), because this might cause unexpected modification to the registers and lead to unsafe code.
6. Move instruction is much cheaper than spill.
7. Code on different threads of the same PU can be different.

Problem Classification

As mentioned in section 1.2, programs executing on different threads can be identical. We call the register allocation problem under such circumstances *Symmetric Register Allocation (**SRA**)*. On the contrary, *Asymmetric Register Allocation (**ARA**)* assumes different programs for different threads. Mixing threads with different computation requirements can achieve better CPU utilization. Since SRA is a sub-problem of ARA, in this paper, we develop our approaches based on ARA. Notice that, although currently most real programs are for SRA, we are not intentionally complicating the problem, because our algorithms are equally necessary and important to SRA, as will be illustrated later, SRA only reduces searching space during inter-thread register allocation, while all techniques in this paper are applicable to both problems. Our goal is develop general techniques that apply without undue restrictions.

Objectives

The number of total available registers is limited. Therefore, in a multithreaded network processor model, we aim to (for ARA) balance the register allocation among all threads, so that more registers are allocated to the thread with higher register pressure and the register allocation is catered to the requirements of different threads in the system. Furthermore, designating a larger number of *shared registers* can help all threads to internally adjust their register pressures without causing spills.

In case there are not enough registers available for all threads, we attempt to split the live ranges inside a thread by using move instructions. Also, our objective is to minimize the number of move instructions inserted. The results show move insertion is cheap and effective.

Problem Formulation

To formalize the problem, we define several concepts.

DEFINITIONS:

PR_i: Number of private registers for thread i, these are physical registers only (*exclusively*) used by thread i.

SR_i: Number of shared registers needed by thread i, these are physical registers used by thread i, but other thread *may* use them as well.

R_i: Number of total physical registers needed by thread i, equals $PR_i + SR_i$

SGR: Number of globally shared registers needed, it is the maximum of shared register demands of each thread, since shared registers can be used by all threads, this is the maximum of all SRs.

N_{reg}: Total number of physical registers available in a PU.

For a thread, PR is the number of physical registers that are exclusively allocated to it or the number of physical registers that can be live across context switch instructions, while SR is the number of allocated physical registers that are dead during context switch, which means they can be shared across threads. For example, in Figure 3.b, for thread 1, $PR_1=1$, $SR_1=2$, for thread 2, $PR_2=0$,$SR_2=1$, therefore, SGR=2.

The relationship and restrictions among these variables are illustrated as the following conditions:

- $SGR = Max(SR_1, SR_2 ... SR_{N_{thd}})$
- $\sum_i PR_i + SGR \le N_{reg}$
- $PR_i + SR_i = R_i$
- For SRA, all PR_i's and SR_i's are equal.

Given these restrictions, we need to assign registers in a way that the overall register need is satisfied and spills are minimized.

3. CONSTRUCTION OF INTERFERENCE GRAPHS

3.1 Non-Switch Region

DEFINITIONS:

Non-Switch Region (NSR): A non-switch region is a maximal connected sub-graph of the CFG without any internal context switch instructions. It contains connected parts from several basic blocks. The boundaries of the NSR are either context switch instructions or program entry/exit points.

Context Switch Boundary (CSB): The program point of the context switch instruction. A CSB separates the basic block it resides, thus becomes the boundary of NSR(s).

A NSR can be constructed by starting from an individual instruction and grown it until all nearby instructions are context switch instruction or program entry/exit points.

To illustrate, Figure 4.a shows the CFG and NSR for a code segment from benchmark "frag" in the Commbench suite [15]. This code segment is from one of the functions to calculate the IP checksum. The CFG consists of 10 basic blocks. Noticeably, there are four context switch instructions, i.e. the read instructions in BB3 and BB7, the explicit ctx_switch instructions in BB5 and BB6. The ctx_switch instructions are inserted by the programmer to avoid the monopoly of the CPU.

Figure 4.b shows the NSRs. After terminating the CFG at the points of context switch instructions (boundaries), we get 3 NSRs. The NSRs are bound by either program entry/exit points or context switch instructions (CSBs). We can assume all terminating are inside basic blocks, therefore some basic blocks are split, like BB5 is split into BB5.a in NSR2 and BB5.b in NSR1. Sometimes, two parts of a separated basic block still belong to the same NSR like the BB7 in Figure 4. For the example in Figure 3, thread 1 has two NSRs, instruction 1 and 2 are in NSR1 and instructions 2 to 12 constitute NSR2. For thread 2, all instructions form one NSR.

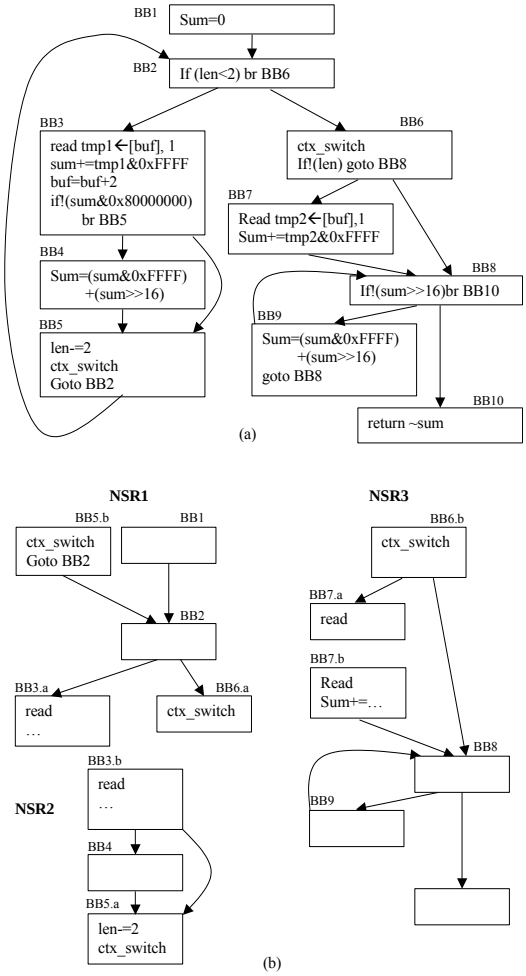

Figure 4. Program CFG and the constructed NSR.

3.2 Interference Graphs

After building the NSR, we build the interference graph, which will guide the register requirement estimation and register allocation. We need to distinguish two kinds of interferences and introduce some other definitions for the interference graph.

DEFINITIONS:

Node: Live range of a virtual register or variable[2]

Boundary Node: Node that is live across the CSB, which may interfere with other boundary nodes.

Internal Node: Node that is not live across CSB.

Boundary Interference: If two boundary nodes are co-live across the same CSB, they are said to be boundary interfering with each other.

Internal Interference: If two nodes (internal or boundary nodes) interfere (co-live at a program point) within a NSR.

Boundary Interference Graph (BIG): A graph consists of all boundary nodes and edges only representing boundary interference.

Internal Interference Graph (IIG): For each NSR, we have an IIG, which only includes the internal nodes live within this NSR and their interference edges.

Global Interference Graph (GIG): The global interference graph includes both boundary nodes and internal nodes. An edge is added if any two nodes (internal or boundary) interfere with each other.

The GIG of the code for the example in Figure 4 is drawn in Figure 5. We assume both len and buf are live at the entry point as the length and the buffer pointer of the packet to be calculated. Also, we assume all variables are dead after their last use in the code. From Figure 4.b, we can see both variable tmp1 and tmp2 are only live within an NSR, so they are internal nodes. Other variables are live across CSB boundaries. They are boundary nodes. For memory read, since all data is first loaded into *transfer registers*[3], the destination register is not assumed to be live across the memory read i.e. the CSB. At BB1, sum, buf and len interfere with each other internally (they also interfere at CSB), thus, the 3 nodes form a clique on the GIG. tmp1 interfere with sum, buf and len in BB3.b, but at the live point of tmp2 in BB7.b, both buf and len are dead. Thus, sum, buf and len form a BIG; the IIG_1 for NSR1 is empty; the IIG_2 for NSR2 includes only tmp1, the IIG_3 for NSR3 includes only tmp2.

Obviously, we have the following claims for each thread.

Claim 1: To avoid spills, the GIG should be colored with R colors and the BIG should be colored with PR colors. Each IIG, as a part of the GIG, should be colored with no more than R colors.

Claim 2: Internal nodes on different IIGs are not connected i.e. they do not interfere with each other.

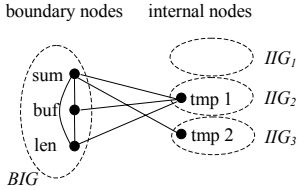

Figure 5. Global interference graph for the example.

Notice that, NSRs and interference graphs can be constructed inter-procedurally. CFGs and NSRs of different functions are connected with edges linking function calls and return points.

[2] Here, we assume each live range represents one variable.

[3] Transfer registers are special registers on IXP used to store data from/to the memory, generally we can assume they are temporary registers dedicated for memory accesses but unavailable as a GPR.

4. OVERALL FRAMEWORK

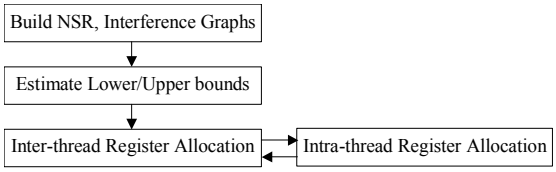

Figure 6. Overall framework.

Figure 6 shows our framework to perform the register allocation. Our first step is to build NSR and interference graphs, we then try to estimate the lower and upper bound of PR and R for each thread. Starting from the upper bound the inter-thread register allocator reduces the overall register requirement gradually until it is within N_{reg}. During this process, when the inter-thread register allocator intends to reduce PR or SR, it calls the intra-thread allocators for all threads. The inter-thread allocator goes towards the direction of the smallest cost increase. The framework allows the intra-thread register allocator to be built separately from the inter-thread register allocator.

5. REGISTER NUMBER ESTIMATION

As the first step towards assigning registers to multiple threads, we need to estimate the number of registers each thread needs based on the interference graph. The estimation helps to guide the distribution of registers to threads at the beginning. Here, we are concerned with finding the bounds for R and PR as defined below. We do not estimate bounds for SR, since the number of SR is always equal to R-PR.

DEFINITIONS:
MinPR, MaxPR: Minimal, maximal number of PR
MinR,MaxR: Minimal, maximal number of R

Lower Bound Estimation

The lower bound is the minimum number of registers a thread needs. First we can get an estimation for the minimum number of private registers (MinPR) one thread needs. A rough estimation is $MinPR \geq \mathbf{RegPCSB_{max}} \equiv Max(\text{number of co-live registers at CSBs})$.

It is obvious that if at a CSB point, there are $RegPCSB_{max}$ nodes (variables) co-live, we need at least this number of private registers since they cannot be shared during context switch. In other words, the minimal number of private registers needed is at least equal to the maximal number of nodes co-live at the CSB boundaries.

The following lemma says this bound can be reached if enough move instructions are inserted. Also, we will explain more about move instruction insertion in Section 7.

Lemma 1: Regardless of shared registers, MinPR can be made equal to $RegPCSB_{max}$ by inserting move instructions.

Proof: If we are given private registers PR_1, PR_2… $PR_{\mathrm{Re}\,gPCSB_{\max}}$, and at a certain CSB, there are V_1, V_2…V_n totally n variables live across, $RegPCSB_{max} \geq n$. Simply, insert n move instructions $PR_1=V_1, PR_2=V_2, \ldots PR_n=V_n$ before the CSB and n move instructions $V_1= PR_1$, $V_2= PR_2$,… $V_n=PR_n$ after the CSB can make the code equivalent to the original and the number of private registers needed is no more than $RegPCSB_{max}$. □

However, in reality, move instruction still costs 1 cycle in our model, although it is much cheaper than spill, we still need to keep the number of inserted move instructions small.

Similarly, we can estimate the MinR needed.

$MinR \geq \mathbf{RegP_{max}} \equiv Max(\#\text{ of co-live registers at program points})$

This lower bound is also achievable given enough move instructions. The proof is similar to the one above.

Upper Bound Estimation

The upper bound gives a maximal number of registers required without any extra move instructions inserted. According to claim 1 in section 3.2, the best estimation for MaxPR and MaxR is the minimal number of colors required to color BIG and GIG. However, for GIG the coloring problem is slightly different from the traditional graph coloring. The problem is to find a coloring scheme for a thread which satisfies:

1. All boundary nodes are colored with at most MaxPR colors
2. All nodes are MaxR colorable
3. Any two interfering nodes are colored differently

For the GIG in Figure 5, all boundary nodes can be minimally colored with 3 colors; thus MaxPR=3. And, all nodes can be minimally colored with 4 colors (there is one 4-node clique), so MaxR=4 ⇒ SR=1.

Actually, there is a tradeoff between MaxPR and MaxR estimation. Reducing MaxPR may induce a larger MaxR. To minimize MaxPR, we can first remove all internal nodes and color the BIG minimally, then insert back the internal nodes and color the graph assuming all boundary nodes have fixed color. To find the tighest (minimal) value of MaxR sufficient to color, we should ignore the condition 1 above, i.e. we could assume that all nodes are indistinguishable and we could simply color the GIG as usual using any coloring allocator. Such a coloring would then minimize MaxR but may give a higher MaxPR.

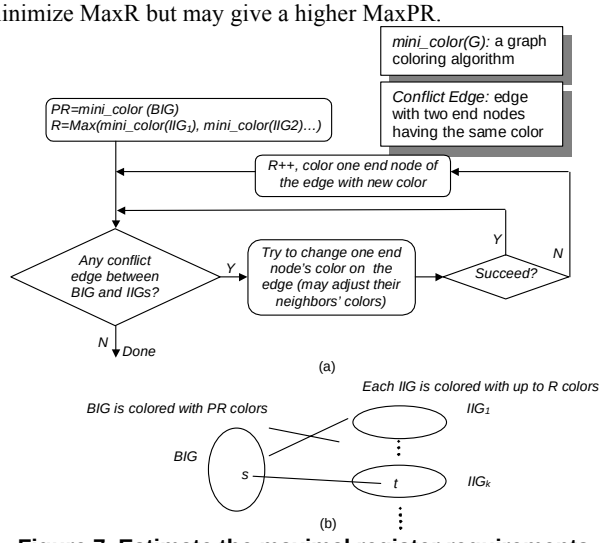

Figure 7. Estimate the maximal register requirements.

We take an approach slightly different from the first one, i.e. we minimize the MaxPR first. This approach is motivated by the fact that increase in PR causes direct increase in total number of registers, while increase in SR only affects the total number of registers when this SR is the maximum among all threads (refer to the formula at the end of section 2). Based on claim 2 mentioned in section 3.2, (i.e. IIGs are not connected with each other) we can color IIGs and BIG separately and then merge them together to keep a tight control on colorability. After merging, edges added between BIG and IIGs may cause conflicts. For example, in Figure

5, when IIGs and BIG are colored separately, variable sum may get the same color as tmp1, leading to color conflict when the edge between them is added during the merge. A general algorithm to color the whole graph altogether may take much more time, since the graph can be big (it includes all live ranges in the program. Some code in our experiment contains hundreds of nodes). Our approach is similar to the fusion-based or region-based register allocation [23], except that our regions are chosen as the IIGs and BIG. The algorithm (Figure 7.a) first builds BIG and IIGs from the GIG and colors each of them independently. In other words, the BIG is colored with color number from 1 to PR, while each IIG is colored with color number from 1 up to R. Some IIGs may be colored with less than R colors, but an IIG can be colored with at most R colors.

The next step tries to merge each IIG with the BIG. The edges between IIG and BIG can cause problem if the two end nodes of an edge have the same color. Such edges are called *Conflict Edges*. The loop in Figure 7.a shows how to resolve all the conflict edges. We illustrate the procedure in Figure 7.b. Suppose boundary node s and internal node t is colored with the same color. If s's color can be changed to another color within color number 1 to PR or t's color can be changed to another color within color number 1 to R, then one of them can be changed to another color to remove this conflict edge. If that fails, we heuristically try to change their neighbors' colors to see if the two nodes can be recolored after that. After all these attempts fail, we have to increase R and t is re-colored with the new color. The algorithm gives MaxPR and MaxR finally. The complexity of the algorithm is $\Sigma O(mini\_color(IIG_i))+O(mini\_color(BIG))+O(\#Edge$ between BIG and IIGs). In contrast, the complexity to color the whole graph is $O(mini\_color(GIG))$. This means the algorithm is also quite fast to try out a given coloring for a thread.

6. INTERTHREAD REGISTER ALLOCATION

6.1 Our approach

One of the difficulties in register allocation for multiple threads is that we do not know exactly how many registers each thread needs. Trying all combinations to find out the best register allocation will cause tremendous amount of compilation time and will be infeasible to build into any practical system.

Our approach is to first get an estimation (range) of how many registers are needed by individual thread via the algorithm proposed in the previous section. From this starting point, we use a greedy heuristic algorithm to approach a sub-optimal solution by reducing the total number of required physical registers gradually. The algorithm also encapsulates the intra-thread register allocator, so that it can be developed independently.

6.2 The Register Allocation Algorithm

After getting the estimated upper bounds $MaxPR_i$ and $MaxR_i$ for each thread, Let $SR_i = MaxR_i - MaxPR_i$ and $PR_i = MaxPR_i$. We can check with the following condition:

$$\sum_i PR_i + Max(S_1, S_2 ... S_{N_{thd}}) \le N_{reg} \quad (**)$$

If this holds, we can assign $SGR = Max(S_1, S_2 ... S_{N_{thd}})$ as the number of globally shared registers and MaxPRi as the number of private registers for each thread to satisfy all register requirements. If the above condition (**) cannot hold good, the register requirement is too high. We must either reduce the PR(s) or SR(s) to satisfy (**).

From (**), we can see, there are two ways to reduce the left side value. Either we can reduce one of the PR_i, which will result in direct reduction of the left-side value. The other way is to reduce SR_i, we should reduce the one(s) with the maximal value. In case multiple SR_i's have the same maximal value, we should consider reducing one of the PR_i if that costs less. The inter-thread register allocation algorithm is shown in Figure 8.

The algorithm first builds GIG and gets the estimations for each thread. If the needed registers are enough (less than N_{reg}), the program simply allocates register and return. Otherwise, it enters a loop to gradually reduce the number of overall register requirement through a greedy algorithm, i.e. every time we choose a direction that can achieve the minimal cost. To reduce the register requirement (i.e. the left side of (**)) by 1, we have many choices. Either we can reduce one of the PRs by 1 or reduce all the maximal SR(s) by 1 to cut down $Max(SR_1, SR_2 ... SR_{N_{thd}})$. Every time we reduce PR of one thread, we check if it is larger than the lower bound. Also, the lower bound of $R_i=PR_i+SR_i>=MinR_i$ is verified when either PR_i or SR_i is reduced.

```
INPUT:  Nthd, Nreg, CFGs of all threads
OUTPUT: all PRi and SRi, SGR, CFGs after register allocation

/*Intra-thread register allocator, returns move cost*/
Intra_thd_allocator(CFG, GIG, PR, SR);

ALGORITHM: Inter_thd_reg_allocation
1.  Build_GIG()
2.
3.  Estimate_reg_requirement()
4.
5.  While(sum(PRi)+max(SR1,SR2…SRNthd)>Nreg)
6.     Foreach PRi>MinPRi and PRi+SRi>MinRi do
7.        cost_PRi=Register allocation cost after reducing PRi by 1.
8.     od
9.
10.    max_SR=max(SR1, SR2…SRNthd)
11.    cost_SR= Register allocation cost after reducing all SRs
12.             that equal max_SR by 1, if all such SRs can
13.             be reduced(by checking PR+SR>MinR)
14.    Find the min one among cost_SR, cost_PR1,cost_PR2…
15.    Choose the one with minimal cost, modify PRs and SRs.
16. Endw
17.
18. Actually modify the CFGs based on new PRs and SRs
19. SGR= Max(SRi)
20. Return all PRi and SRi, SGR, all CFGs
```

Figure 8. Algorithm for inter-thread register allocation.

The function *Intra_thd_allocator* is an intra-thread register allocator. It accepts the PR and SR, then tries to return an allocation using PR and SR number of registers. This function is called when we calculate register allocation cost for each thread and when we finally modify the CFGs. It returns the allocation cost. Actually, the interference graph and coloring scheme given by the function *Estimate_reg_requirement* can be passed to the intra-thread register allocator as a starting point. However, to provide more flexibility, we leave this to the implementation of *Intra_thd_allocacor*.

The complexity of our heuristic algorithm is $O(N_{reg}*N_{thd})*$ $O(Intra\_thd\_allocator)$, which largely depends on the complexity of the intra-thread register allocator. Our register allocation algorithm generates satisfactory solution for all benchmark programs within almost negligible compilation time.

7. INTRATHREAD REGISTER ALLOCATION

The intra-thread register allocator attempts to allocate up to

PR number of physical registers to boundary nodes and up to R=PR+SR physical registers to all nodes.

7.1 Move Insertion and Live Range Splitting

Our intra-register allocation is based on live range splitting and move instruction insertion. Live range splitting has been used in register allocation [11] to spill part of the live range to memory. In this paper, we attempt to split the live ranges by inserting move instructions to reduce the chromatic number. Lemma 1 has shown that through live range splitting MinPR can be reached. Figure 9 gives another example. In Figure 9.a, live ranges A B and C interfere with each other at three different CSB points. The lower bound lemma in section 3 gives MinPR=2, but the interference graph must be colored with 3 colors, because A,B, and C form a clique. In Figure 9.b we split the live range of variable A into A_1 and A_2 by inserting move instruction at the split point. The resulting interference graph can be colored with 2 colors which is equal to MinPR. Notice that, this is also the way we reduce the number of registers required in the first example (Figure 3.c).

In our intra-thread allocation algorithm, we focus on live range splitting through move insertion because spill is too expensive on network processor and our experiments show MinPR (MinR) is much smaller than MaxPR (MaxR). This provides us room to reduce chromatic number towards the lower bound by inserting move instructions.

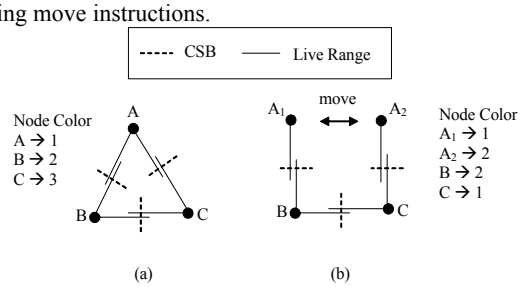

Figure 9. Live range splitting via move insertion.

7.2 Intra-thread Register Allocation Algorithm

Our register allocator works incrementally, i.e. it records the *context* (interference graph with split nodes and the position of move instructions) of the last 2 invocations and modifies the context to satisfy the new PR and SR values. Notice that the inter-thread allocation algorithm in Figure 8 calls *Intra_thd_allocator* multiple times. In each step, either it accepts the previous context and reduces PR or SR by 1 or it rejects the previous modification and starts from the previous to previous context and reduces PR or SR by 1. Incremental modification can save time for otherwise repetitive work. Further, based on the records of the two contexts, we can assume that each time the allocator is invoked, it attempts to reduce either PR or SR by 1 from one of the recorded contexts. We name these two kinds of invocation as *Reduce-PR invocation* and *Reduce-SR invocation*.

Reduce-PR Invocation

In this type of invocation the allocator wants to reduce the PR by one from its last invocation. In other words, the last accepted context can color all boundary nodes with PR colors and this invocation wants to color it with PR-1 colors.

In this stage, we assume all move instructions are inserted near the CSB. With this assumption, we do not need to alter the colors of internal nodes. Normally, changing the color of both internal and boundary nodes might induce more move instructions (in this case we must split the live range to recolor an internal nodes) and increase the cost accordingly. Later, we will show some of the move instructions at the CSB can be eliminated by merging them with move instructions inside the NSR. This actually relocates the move instructions from the CSB boundary. Before the discussion of our algorithm, we first define *Neighbor Color Number (NCN)*.

Definition:
Neighbor Color Number (NCN): The number of colors used by the neighbors of a given node in a colored graph.

```
INPUT:  PR, SR
OUTPUT: cost (number of inserted move instructions)

Static context_pre, context_pre_pre

1.  FUNCTION Reduce_PR(context):cost
2.  Begin
3.    Foreach color c in PR do
4.      Cost=0
5.      Foreach node t in Set_color_node(c,BIG) do
6.        If NCN(t,BIG)<PR-1 then
7.          Change t to another color c' in PR other than c.
8.          Cost+=min(Cut_if_conflict(t,c,c')) for all possible c'
9.        Else
10.         Cost+=min(NSR_exclusion_cost(t,c,c')) for each
11.                 color c' in PR other than c
12.         Add newly split node with color c to Set_color_node(c,BIG)
13.                 if it is boundary node
14.       Endif
15.     od
16.     Eliminate_unnecessary_move()
17.     Record to min_cost if this cost is smaller and record the context.
18.   od
19.   Keep the minimal cost context and return min_cost
20. End

21. FUNCTION Reduce_SR(context):cost
22. Begin
23.   Foreach color c in SR
24.     Cost=0
25.     Foreach NSR_i color c is used do
26.       Foreach internal node t in Set_color_node(c,IIG_i) do
27.         If NCN(t, GIG)<R-1 then
28.           Color t with a color other than c.
29.         Else
30.           Cost+=min(live_range_exclusion_cost(t,c,c'))
31.                   For each color c' in R other than c
32.           Add newly split node with color c to Set_color_node(c,IIGi)
33.         Endif
34.       od
35.     od
36.     Eliminate_unnecessary_move()
37.     Record to min_cost if this cost is smaller and record the context.
38.   od
39.   Keep the minimal cost context and return min_cost
40. End

41. FUNCTION Intra_thd_allocator(PR,SR):cost
42. Begin
43.   According to the accepted context, pick stored either context_pre
44.     or context_pre_pre => context.
45.   If(PR is reduced) return Reduce_PR(context)
46.   Else if (SR is reduced) return Reduce_SR(context)
47.   Else return cost for the context //no change
48. End
```

Figure 10. Algorithm for intra-thread register allocation.

The algorithm in Figure 10 uses function *NCN(t,BIG)* to get the neighbor color number of node t on the BIG. The algorithm also works in a greedy manner. It tries each color c in PR colors and checks the cost to eliminate that color. Then, the color with least elimination cost is selected to be eliminated and all needed move instructions are inserted. Function *Set_color_node(c,BIG)* returns the set of nodes on BIG with color c. We need to change every node in this set to a different color in PR.

Firstly, we check the NCN of t that has color c on the BIG. If this number is less than PR-1 (which means there is at least one color available in PR not used by its neighbors), we can change t to another color. Since we have changed t's color on BIG and t may internally interfere with other internal nodes or boundary nodes (two boundary nodes can interfere only inside NSR but not on the CSB), we need to check if there is a color conflict. The function *Cut_if_conflict(t,c,c')* attempts to insert move instructions to disconnect such edges. Figure 11 shows how the disconnection is done and the corresponding changes on the GIG. In Figure 11.a, s is originally colored with color c'; after node t is changed to color c' from color c it conflicts with internal node s. We insert a move at the CSB, so live range t is split. The part of the live range t in NSR2 becomes t', and this part can keep color c, so it does not conflict with s, while, on the BIG, t is changed to color c'. Figure 11.b shows the changes on the GIG. The edge between t and s gets eliminated after t' splits from t. t' keeps the original color of t, so in the IIG, it is compatible with s, while on the BIG, the color of t is changed. In the algorithm, we try every candidate color for t and pick the one with minimal cost.

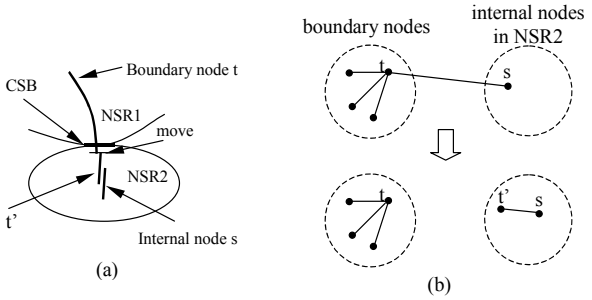

Figure 11. Node splitting to change the color of node t.

If this step fails, i.e. *NCN(t ,BIG)*=PR-1, the algorithm calls function *NSR_exclusion_cost(t,c,c')* to get the cost of changing t to another color c' and to exclude all the NSRs with conflict nodes. *NSR_exclusion_cost* looks at each NSR where t is live to see if there is any node with color c' in it. If so, the NSR is excluded by splitting the live range of t in that NSR and by inserting move instructions. In our approach, the NSRs are split in whole, i.e. either the live range in that NSR is kept with color c' (if no conflict) or the live range is split (after splitting, t' in that NSR keeps color c).

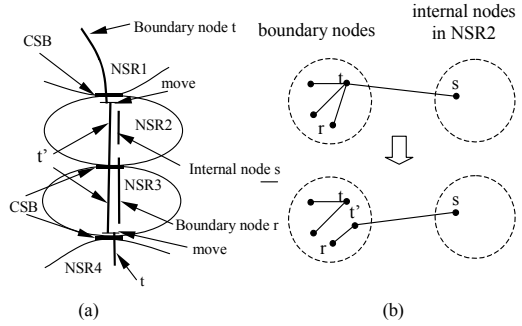

Figure 12. NSR exclusion to reduce PR.

Figure 12 shows how NSR exclusion is done. Boundary node t cannot change to color c' because the boundary node r and the internal node s are using color c'. The conflict NSRs are NSR2 and NSR3, where s and r are live. So, these two NSRs are excluded from the live range of the original boundary node t. On the GIG, we see t' is split from t and t now can be colored with c'. t' keeps color c and it is still compatible with s and r. Notice that, after splitting, the edge originally connected from r to t is connected to t'. Therefore, the NCN of t is reduced and t can be recolored with c'.

The algorithm tries each color other than c to recolor t and finds the minimal value to finally color t. Also notice that, after this step, t' is colored with c and, if it is a boundary node, we should add t' to *Set_color_node(c,BIG)* and we will color it with some other color during the later iterations. *Set_color_node(c,BIG)* will not increase infinitely, since further splitting t' will finally generate internal nodes.

Reduce-SR Invocation

To reduce SR, we check with each color c in SR to see which one should be reduced with minimal cost. The cost is calculated by adding up costs in every NSR where this color is used. Also notice that in this step, all boundary nodes are assumed to have fixed colors so that the phase will not affect the PR number.

The algorithm tries to recolor node with color c in a NSR to other color. If the node on the GIG has NCN less than R-1, we can just pick that color and color the node without any cost. Otherwise, live range splitting is needed.

Live range splitting is illustrated in Figure 13. In Figure 13.a, the example has 3 basic blocks. Live range t is recolored with color c', however, live range s also uses color c'. Our algorithm then splits t at the boundary where the two live ranges overlap. After splitting, t' can still use color c and t now changes to c'. We assign the color with minimal cost to node t. After the splitting, node t' is push into *Set_color_node(c,IIG$_i$)*, because now it bears color c.

This process will finally stop. After each splitting, the live range with color c is reduced. Since the value R-1 $\geq$ RegP$_{max}$ (according to the lower bound estimation in section 5 and the algorithm in Figure 7), in the extreme case, each live range is a single program point, there will be at most RegP$_{max}$ nodes co-live and live range with color c can always be recolored.

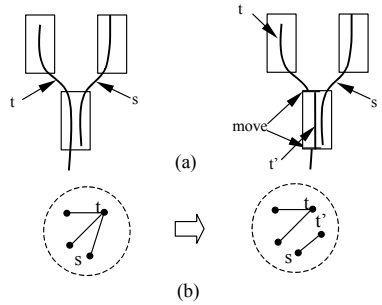

Figure 13. Excluding a live range within NSR to reduce SR.

Eliminate Unnecessary Moves

During the attempt to reduce PR, we assume that all move instructions are inserted near the CSB boundary and during reduce SR, some move instructions are inserted inside the NSR. At this point, we can merge some of the internal move instructions with those at the boundary. For two consecutive moves, the first move instruction to the live range is unnecessary if the color at the entrance to the first move is also acceptable in the region between the two move instructions. We can safely eliminate the first move and this actually relaxes the restriction in Reduce_PR to bind moves to the CSB.

8. THE SRA PROBLEM

For SRA problem (defined in section section 2), given the

PRs are equal and SRs are also equal. The restriction can be rewritten in a simple form:

$$N_{thd} * PR + SR \leq N_{reg}$$

Thus, the inter-thread register allocation algorithm can also be simplified. There are only two possibilities to reduce the register requirements. Due to the shrunk solution space, for algorithm in Figure 8, we can actually traverse all the possible PRs and SRs to find the best solution.

9. EXPERIMENTAL RESULTS

The evaluation of our algorithm is done with the Intel-provided simulation environment—IXP1200 Developer Benchmark 2.01. The IXP1200 workbench supports *cycle-accurate* simulation for IXP microengines and other peripheries with high fidelity.

In this section, we experiment with 11 benchmark programs and some of their combinations to see the effectiveness of the register allocator. These benchmarks are collected from Commbench[15], Netbench[16], Intel provided example code and a packet scheduling algorithm from [18]. To evaluate our algorithm, the benchmark programs are rewritten in IXP C code (a subset of standard C) and a few of them are directly written in assembly (microcode). For those written in assembly code, we restore the virtual registers so that our register allocator can work on the live ranges from scratch. Our pass builds the CFG and interference graph from the assembly code, after simple translation of the assembly directives. The assembly code is then passed to the assembler to generate machine code. The IXP assembly consists of only 40 RISC instructions which makes the translation easy. The assembler simply exits if too many registers are required. However, after our pass, the register requirements are always satisfied, so the machine code can be generated properly. Table 1 shows the properties of the benchmark programs. The code size is number of instructions after code generation. The cycle counts are measure as follows: for some programs like L2l3forward, it cannot run to a stop in finite time, since these programs all runs in a while loop to accept and process packets, the cycle counts are averaged number per iteration of the main loop. We list ***CTX instructions*** (context switch instructions, which includes load/store, voluntary context switch and other I/O operations that can cause context switch) each benchmark has. Roughly, about 10% instructions are CTX instructions. The CTX instructions here do not include spill instructions, as we have removed all spills and reconstructed original live ranges (we did this based on the source code and the annotations embedded in the generated assembly code by the Intel IXP compiler). The number of live ranges (nodes on the GIG) is listed in the 5th column. These numbers come from the restored virtual registers. Column 6 and 7 are maximal register pressures in the program ($RegP_{max}$) and maximal register pressure at the CSBs ($RegPCSB_{max}$). These are the lower bound estimation for register requirements of the threads. Column 8 and 9 are the upper bound estimation for R and PR based on the algorithm in Figure 7. The 10th and 11th column give statistics for the numbers of NSRs and their average sizes. One observation is that normally larger NSR leads to bigger difference between the maximal and minimal value of P and PR. Because more internal nodes can exist in larger NSRs, the register pressure for GIG should exceed the BIG with larger margin.

Figure 14 evaluates our inter-thread register allocation algorithm for SRA. The same evaluation for ARA is combined in Table 3. For each benchmark program, we show two relevant bars. The first bar is the number of registers allocated to the benchmark assuming only single thread is available. We use a Chaitin [9] style register allocator for comparison with our shared register allocator. The second and third bars are the number of private registers and shared registers assigned with our inter-thread register allocation algorithm. The same benchmark is assumed to execute on four threads. The algorithm continues until the cost returned is non-zero, which means we want to test how many PR and SRs are needed without any move instruction insertions with the inter-thread allocation algorithm. The figure shows that the number of private registers allocated for the multi-threaded case is less than the number of registers needed for standalone register allocation. This is not surprising because shared registers can take care the higher register pressure inside the NSRs. If no shared registers are used and each thread runs the single-thread register allocator, many registers are wasted. Compared to the case with multi-threaded register requirements i.e. 4*PR+SR, the average total register saving for all benchmarks is 24%.

In Table 2, we collect data for the extreme case with our register allocation algorithm, i.e. the maximal number of move instructions that will be inserted, if only the minimal number of registers is allocated. This means our algorithm must split many live ranges to reach the minimal number of registers. The move insertion overhead in the extreme case is mostly within 10% of the total number of instructions for the benchmarks. This cost is affordable compared to the overhead due to register spill if the register number is out of range with the single thread register allocation algorithm.

Finally, Table 3 evaluates our register allocation algorithm for ARA with 3 scenarios. Notice that all tasks are periodic, independently sharing the CPU and execute forever. Thus, we measure the performance improvements of each thread in terms of the percentage reduction of cycles per iteration. The first scenario put two Md5 programs on thread 0 and 1, two fir2dim on thread 2 and 3. This can be a processing module between the receiving and sending module. Our data show the PR and SR assigned, the number of live ranges after the register allocation (#Live Ranges), context switch instruction number reduction and cycle change. The column of "#CTX Reg Spill" is the original code generated by the Intel compiler that allocates registers with spilling and without register sharing across threads (only allocate 32 registers for each thread). And, "#CTX Reg Sharing" is the number with our allocator (actually no change compared with Table 1, because we avoid spills). The same is true for cycle count ("#Cycle Reg Spill" and "#Cycle Reg Sharing"). The fir2dim actually runs slower due to inserted moves. But this is profitable due to the big saving from Md5. Thus, the allocator is able to boost the performance critical thread (Md5) by slightly slowing down less performance critical one (fir2dim). The second scenario consists of L2l3fwd receive and send on thread 0 and 1 and Md5 on thread 2 and 3. This can be a complete processing modules serving on one sending and one receiving port. The results still show the spills are saved for Md5 with minor costs for moves on L2l3fwd threads. The last scenario runs wraps receive and send on thread 0 and 1, fir2dim and frag on thread 2 and 3. The allocator balances register allocation to satisfy wraps thread. Due to a high register pressure, wraps receive and send can run much slower (due to spills) if registers are not allocated properly. Our results show that over 20% speedup is achieved for wraps, whereas only slight slowdown is incurred for the other two benchmarks, which is in accordance with our optimization objective of boosting performance critical thread.

10. RELATED WORK

The multi-threaded architecture of our model (similar to IXP) differs from traditional general purpose multi-threaded processors or SMT processors in that the number of threads and the code for each thread is known beforehand (at compile time). Further, the architecture exposes context switch to users (actually the programmer should handle everything except the save/restore of PC for each thread). This makes register sharing across threads difficult, since registers are not saved during context switch. On the other hand, exposed architecture features allow the compiler to undertake inter-thread register allocation which is the subject of this paper.

This problem is also different from the traditional concepts of caller-save and callee-save registers, since registers cannot be saved to the memory during context switch due to the high costs of memory operations. The only chance to use a register that is also used (shared) by other threads is to guarantee the register is dead during the context switch. [24] studies live range analyses for context switches at the procedure boundary on Alpha machines. Optimizations are conducted to minimize the number of registers that should be saved during context switches—in [24] it is equivalent to reduce the number of callee-save registers. In contrast, context switches taking place on the network processor are more frequent (reaching basic block level) thus require analyses at finer granularities; it is not profitable to save/restore but allow small deadness at context switch point for a fine granularity allocation.

[20] talks about the inter-task register allocation problem for embedded systems with static OS and predetermined tasks. The goal of the paper is to minimize the number of registers that should be saved during context switch. However, the assumption that tasks have fixed priority and saving the variables to memory during context switch is not applicable in our model. Finally, their assumption that tasks can be preempted at any given program points except for critical sections is not true for our network processor model either. Therefore, the techniques proposed in their work are not applicable to our problem.

Table 1. Benchmark applications.

	Code Size	Cycle/ iteration	#CTX insns w/o spills	#live ranges	$RegP_{max}$	$RegPCSB_{max}$	MaxR	MaxPR	#NSR	Ave.NSR Size
Crc	78	52280	6	14	6	5	9	8	4	19.5
Drr	108	207037	12	11	5	4	8	6	7	15.43
Fir2dim	447	159149	32	36	21	17	27	20	19	23.53
Frag	271	28620	30	26	16	12	22	18	18	15.06
Kmp	123	148059	12	13	7	5	10	7	5	24.6
L2l3fwd (Rec)	635	1253.34	55	131	30	28	35	34	28	22.67
L2l3fwd (Send)	690	721.28	88	115	24	21	31	25	33	20.91
Lzw	126	43163	12	18	13	10	15	11	9	14
Md5	913	3983292	56	142	41	37	60	46	31	29.45
Wraps(receive)	875	2048.37	85	145	45	39	59	47	32	27.34
Wraps(send)	921	1264.87	103	135	49	40	65	50	37	24.89

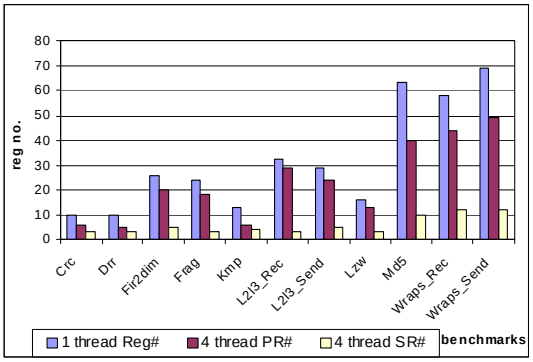

Figure 14. Original vs. SRA register allocation.

Table 2. Minimal case move insertion.

	PR	SR	# Move
Crc	5	1	10
Drr	4	1	11
Fir2dim	17	4	19
Frag	12	4	15
Kmp	5	2	8
L2l3_Rec	28	2	23
L2l3_Send	21	3	30
Lzw	10	3	7
Md5	37	4	45
Wraps(receive)	39	6	47
Wraps(send)	40	9	50

Table 3. Static and dynamic results for different ARA scenarios.

	Benchmark (Thread#)	PR	SR	#Live Ranges	#Move Inserted	#CTX Reg Spill	#CTX Reg. sharing	#CTX Reduction	#Cycle Reg Spill	#Cycle Reg. sharing	#Cycle Reduction
Scenario 1	Md5(0,1)	41	10	152	20	79	56	29.11%	5028375	4039294	19.67%
	Fir2dim(2,3)	18	10	38	4	32	32	0	159149	163515	-2.74%
Scenario 2	L2l3fwd_rec(0)	28	5	133	24	55	55	0	1253.34	1273.06	-1.57%
	L2l3fwd_send(1)	21	5	120	18	88	88	0	721.28	749.18	-3.87%
	Md5(2,3)	37	5	165	38	79	56	29.11%	5028375	4113871	18.19%
Scenario 3	Wraps_rec(0)	42	11	161	40	123	85	30.89%	2773.13	2134.52	23.03%
	Wraps_send(1)	44	11	153	36	141	103	26.95%	1709.36	1334.71	21.92%
	Fir2dim(2)	17	11	40	6	32	32	0	159149	164201	-3.17%
	Frag(3)	14	11	28	4	30	30	0	28620	28979	-1.25%

A recent publication [19] studies register allocation problem for single thread on the IXP network processor. The compiler is dedicated to the particular processor with consideration to many architecture details which involve irregularities but they do not focus on an important IXP feature--multi-threading. The goal of our paper is to study a generalized model for multi-threaded register allocation, so it can be extended to other network processors with similar designs.

We focus on the balancing of registers among different threads and the allocation of shared registers to meet the overall register demands across threads. We show that the problem is quite involved and provide a systematic solution to balance register requirements across threads by determining the number of registers to be shared and by splitting live ranges within selected threads inserting moves minimally.

11. CONCLUSION

In conclusion, our approach attempts to maximize sharing of registers across threads to make more registers available to them reducing their spills. The values that are not live across the context switch program points are held in shared registers. Maximizing shared registers in turn reduces the spill and context switches making it safer to keep more ranges in shared registers. We approach this problem from zero-spill accounting only for mandatory load/stores and other context switches and work out an approach to balance the registers across threads stabilizing the solution as above.

The results show that we are able to minimize register requirements in SRA setting and are able to improve the cycle counts substantially in the ARA setting for large benchmarks executing on different threads. This means that it is viable to develop multi-threaded large applications on IXP effectively with a good compiler support. The solution is able to speed-up performance critical threads by meeting their demands through maximal sharing of registers.

12. ACKNOWLEDGEMENT

This work was supported in part by NSF grants CCR-0220262, CCR-0208953 and CCR-03263.

13. REFERENCES

[1] Feliks J. Welfeld "Network processing in content inspection applications," *In Proceedings of International Symposium on Systems Synthesis*, Sep. 2001.

[2] T.Spalink, S.Karlin, L.Peterson, Y.Gottlieb, "Building a Robust Software-Based Router Using Network Processors," *In Proceedings of ACM Symposium on Operating Systems Principles*, Oct. 2001.

[3] J. Wagner and R. Leupers, "C Compiler Design for an Industrial Network Processor," *In Proceedings of ACM SIGPLAN Conference on Languages, Compiler, and Tools for Embedded Systems*, Jun. 2001.

[4] J.Kim, S.Jung, Y.Park, "Experience with a Retargetable Compiler for a Commercial Network Processor," *In Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, Oct. 2002.

[5] J. Liu, T. Kong, and F. Chow. "Effective compilation support for variable instruction set architecture", *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques,* Sep. 2002.

[6] T.H.Cormen, C.E.Leiserson, R.L.Rivest, *Introduction to algorithms*, MIT Press, 1989

[7] C.H.Papadimitriou, K.Steiglitz, *Combinatorial optimization Algorithms and Complexity*, Dover Publications INC, 1998.

[8] A.V.Aho, R.Sethi, J.D.Ullman, *Compilers Principles, Techniques and Tools*, Addison-Wesley, Reading, MA, 1986

[9] S.S.Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufman, 1997.

[10] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems,* Jul. 1987.

[11] Fred C. Chow and John L. Hennessy, "The priority-based coloring approach to register allocation", *ACM Transactions on Programming Languages and Systems*, Oct. 1990.

[12] L. George and A. Appel. "Iterated Register Coalescing," *ACM Transactions on Programming Languages and Systems*, May 1996.

[13] G.J. Chaitin, "Register allocation and spilling via graph coloring", *In Proceedings of International Conference on Compiler Construction*, Jun. 1982.

[14] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, P. Markstein, "Register allocation via coloring", *Computer Language*. Vol.6, pp.47--57, Jan. 1981.

[15] T. Wolf and M. Franklin, "CommBench – A Telecommunication Benchmark for Network Processors", *In Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, Apr. 2000.

[16] G.Memik, W.H.Mangione-Smith, W. Hu., "NetBench: A Benchmarking Suite for Network Processors", *In Proceedings of the International Conference on Computer Aided Design*, Nov. 2001.

[17] Rajiv Gupta, Mary Lou Soffa, and Tim Steele, "Register allocation via clique separators", *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1989.

[18] Xiaotong Zhuang, Jian Liu, "WRAPS Scheduling and Its Efficient Implementation on Network Processors", *In Proceedings of the 9th International Conference on High Performance Computing*, Dec. 2002.

[19] L. George, M. Blume, "Taming the IXP Network Processor", *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2003.

[20] V. Barthelmann, "Inter-Task Register-Allocation for Static Operating Systems", *In Proceedings of ACM SIGPLAN Conference on Languages, Compiler, and Tools for Embedded Systems*, Jun. 2002.

[21] http://www.cs.purdue.edu/np/npc.html

[22] Timothy Sherwood, George Varghese, and Brad Calder, "A Pipelined Memory Architecture for High Throughput Network Processors," *In Proceedings of the 30th Annual Intl. Symposium on Computer Architecture*, Jun. 2003.

[23] David Callahan, Brian Koblenz, "Register allocation via hierarchical graph coloring," *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun. 1991.

[24] Dirk Grunwald, Rich Neves, "Whole-Program Optimization for Time and Space Efficient Threads," *In Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.