System Level Methodology for Programming CMP based Multi-threaded Network Processor Architectures*

Vijaykumar Ramamurthi, Jason McCollum, Christopher Ostler, and Karam S. Chatha,
Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85281.
Email: kchatha@asu.edu

Abstract

The increasing demand for programmable platforms that enable high bandwidth communication traffic processing has led to the advent of chip multi-processor (CMP) based multi-threaded network processor (NP) architectures. The CMP based architectures include a multitude of heterogeneous memory units ranging from on-chip register banks, local data memories, and scratch pads to multiple banks of off-chip SRAM and DRAM. Implementation of applications on such complex CMP architectures involves mapping of functionality on processing units, and mapping of data items on the memory units with an objective of maximizing the throughput. This paper presents a system-level methodology that consists of a programming model and optimization techniques for solving the functionality and memory mapping problem on CMP based multi-threaded NP architectures. The proposed techniques are evaluated by implementing three representative NP applications on the Intel IXP2400 processor which belongs to the class of CMP based multi-threaded architectures.

1 Introduction

As the traffic processing needs of modern communication networks continue to increase exponentially, network processing has become a major bottleneck. Programmable application specific network processor (NP) architectures are seen as a viable and cost effective solution to this problem [1]. In these architectures, NP designers have employed a large variety of hardware techniques to accelerate packet processing, including parallel processing by introducing multi-processor architectures with support for multi-threading, special-purpose hardware, memory architectures, on-chip communication mechanisms, and the use of peripherals [2]. Despite the architectural innovations, very little effort has been made in making these architectures easily programmable. The complex architecture of the NP and the difficulty of programming it [3] motivate the need for a well-defined programming methodology for application development on these platforms.

Currently, NPs are programmed using assembly language or a subset of C prohibiting the scaling of performance in a significant way. Any changes to the underlying architecture will increase the development costs, with the exception of increasing clock frequency and increasing the number of packet processing engines. Also, the developer still must divide the functionality among the threads and processors, as well as determine how best to utilize memory elements in order to obtain optimum performance. This low-level approach to programming places a large burden on the developer, requiring a detailed understanding of the architecture in order to implement any packet processing application. Additionally, it is an extremely time consuming process. This represents a significant gap between the application domain and the underlying architecture. To close this gap, we need a programmers model combined with a well defined design methodology to efficiently map and optimize a network processing application on the architectural features of the underlying multiprocessor platform.

Existing work in the area of task allocation [4] and system-level design [5] propose generalized solutions that are not suitable for NP architectures. In particular, these schemes do not take into account thread and storage limitations, which are critical factors that affect the quality of the mapping to multi-threaded architectures. NP-Click[6] presents a programming model for the IXP1200 NP. They provide an abstraction layer for programming purposes but do not address the issue of thread boundaries or data layout. Work presented in [7] focused only on the task allocation problem in Intel IXP2400 architecture and did not address the data mapping problem. In a CMP based multi-threaded architecture the data memory (register, local memory) in a PE is divided among the various threads and therefore is an important design constraint. If the memory is not allocated in an appropriate manner, then the desired number of threads cannot be executed and maximum throughput is not realized.

* The research presented in this paper was supported by a grant provided by the Consortium for Embedded Systems, Tempe, AZ, 85287-9009.

Proceedings of the IEEE Computer Society Annual Symposium on VLSI
New Frontiers in VLSI Design
0-7695-2365-X/05 $20.00 © 2005 IEEE

As designers increasingly adopt programmable platforms, we believe a strong programming methodology will be a key to harnessing the power of these new architectures. This paper presents a programming model and a system-level programming methodology which will effectively address the current design gap. The programming model utilizes a process graph for representing the target application, which is well suited for implementation on multi-processor architectures. We then apply a set of transformations and optimizations on the model, based on the underlying architecture, to map the application functionality and data with an objective of maximizing the throughput. Our methodology enables the designer to make intelligent choices in the design phase itself by providing performance estimates. Our methodology also eliminates the highly time-consuming step of implementing the application on the platform and then optimizing for the required performance.

The paper is organized as follows: Section 2 gives background on Intel IXP2400 architecture, Section 3 discusses related work, Section 4 describes the design methodology in detail, Section 5 presents our experiments, and finally Section 6 concludes the paper.

2 Intel IXP2400 architecture

This section provides a detailed architectural overview of the Intel IXP2400 processor which belongs to the class of CMP based multi-threaded architectures [8] . The process of mapping an application to the IXP2400 consists of determining the best allocation of tasks and memory. Thus, although the platform offers many additional features, we will focus mainly on the capabilities of the micro-engines and the available memory interfaces.

The IXP2400 architecture combines an Intel XScale processor core with eight 32-bit independent multi-threaded micro-engines, grouped into two clusters as shown in Figure 1. The micro-engines are based on Intel's second-generation multi-threaded RISC processors. Each thread (context) in a micro-engine has its own register set, program counter, and controller specific local registers. Fast context swapping allows another context to do computation while the first context waits for an I/O operation. Each thread can be in one of four different states i) Inactive: used if the application does not want to use all threads, ii) Ready: the thread is ready to execute, iii) Execute: this is the executing state; a thread stays in this state until an instruction causes it to change states or a context swap is made, and iv) Sleep: the thread is waiting for external events to occur. Only one context can be in the executing state at any given time. Each micro-engine contains 640 32-bit words of local memory. This memory is divided equally among the threads on the micro-engine. A micro-engine has 256 32-bit general purpose registers, used under program control to supply operands to the execution data path.

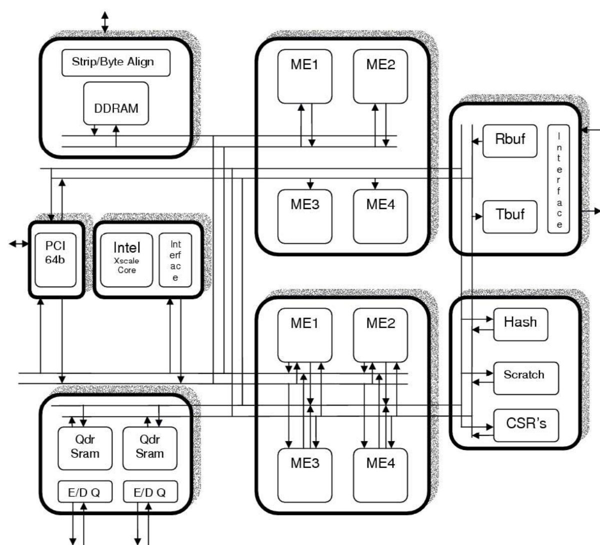

Figure 1. IXP2400 Processor Architecture

Each micro-engine has access to three types of external memory: scratch pad, SRAM and DRAM. Scratch pad is an on-chip memory while SRAM and DRAM are off-chip. The IXP2400 is equipped with 16KB of scratch pad memory that acts a fast shared memory accessible from all the micro-engines. The IXP2400 provides two channels of QDR SRAM. The SRAM controller has hardware support for queue, linked list, and ring operations that can be utilized for implementation of FIFOs. It also supports synchronization operations like atomic bit operations. The IXP2400 has one channel of DRAM that supports up to 2 GB. It is primarily used to buffer incoming packets as it has a fast interface with the media switch fabric that is responsible for receiving/transmitting the packets from the external environment.

3 System-level Design Methodology

The proposed methodology for mapping applications onto CMP based multi-threaded network processors is shown in Figure 2. The key architectural features of the target architecture that are relevant for the mapping purposes are specified via an intermediate format by the programmer. The application itself is specified by a process network based model. The process model is characterized by test streams and an intermediate format of the application is derived for computation and data mapping. This intermediate format is operated upon by our process and data optimization techniques with an objective of maximizing the throughput of the application. In the following sections we explain each of the design stages in detail.

Proceedings of the IEEE Computer Society Annual Symposium on VLSI
New Frontiers in VLSI Design
0-7695-2365-X/05 $20.00 © 2005 **IEEE**

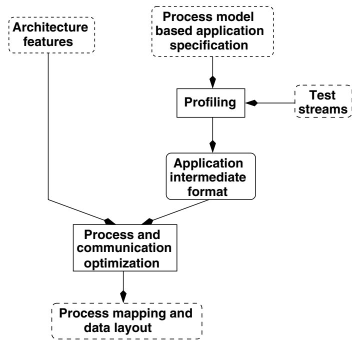

Figure 2. Overview of methodology

3.1 Preliminaries

CMP architecture features : Our methodology assumes a CMP based multi-threaded architecture very similar to the Intel IXP 2400 processor. The architecture is specified by a tuple $\mathcal{A}\langle\mathcal{P},\mathcal{M}\rangle$ where $\mathcal{P}$ denotes the set of PE and $\mathcal{M}$ denotes the set of memory elements (external to each PE) that are available to the programmer.

Each PE $c_i \in \mathcal{P}$ is given by another tuple $p\langle\kappa,\sigma_r,\sigma_d,\omega_d,\sigma_c\rangle$ where $\kappa(c_i)$ denotes the maximum number of threads that can be supported on the PE, $\sigma_r(p_i)$ denotes the total number of registers in the PE, $\sigma_d(p_i)$ denotes the size of local data memory of the PE, $\omega_d(p_i)$ denotes the maximum bandwidth for the local data memory of p_i, and $\sigma_c(p_i)$ denotes the size of the code memory of the PE. Without the loss of generality we assume that all the PE have identical architectural features.

Each memory element $m \in \mathcal{M}(\mathcal{A})$ that is external to the PE is given by a tuple $m\langle\sigma,\omega\rangle$ where σ and ω denote similar quantities as defined above.

Process model : The application is specified in the form of a process network that consists of concurrently executing processes. The inter-process communication takes place through bounded point-to-point FIFO and shared memories. The bounded FIFO support blocking read (on empty) and blocking write (on full) operations. The shared memory in addition to standard read and write operations also supports exclusive read and write operations with implicit mutex semaphores. We place an additional constraint that the programmer cannot dynamically allocate and de-allocate memory during the run-time of the process. The programmer can statically allocate the dynamic memory during an initialization phase, and then the memory usage remains constant for the lifetime of the application.

Application characterization and intermediate format : The process model based application specification is profiled to obtain timing characteristics for each process and the access frequency for the various data items. Each process is characterized separately by utilizing designer specified test streams. In order to obtain the timing characteristics of a process, we apply a preliminary data mapping. The motivation of the preliminary data mapping is two fold. In the NPs on-chip data memory is a precious resource. Hence, in the preliminary mapping data item are mapped to off-chip memory wherever applicable. Further, and more importantly the local memory of the PE is typically divided equally among all threads. If there is insufficient memory, it would result in fewer threads and consequently lower throughput. Therefore, data mapping to local data memory is limited to only scalars. The preliminary data mapping is as follows:

- Map all process variables that are scalar to the local memory within a PE.
- Map all the process variables that are aggregate, FIFO and shared memories to an off-chip memory. If there are multiple off-chip memories map the data items to the slowest memory.

Some NPs support fast interfaces between an off-chip memory and the media access layer. If the target architecture contains such an interface the incoming and outgoing packets are mapped to the respective off-chip memory.

The characterized application specification is then captured in an intermediate format given by a tuple $\mathcal{N}\langle\mathcal{V},\mathcal{F},\mathcal{S}\rangle$ where $\mathcal{V}$ is the set of processes, $\mathcal{F}$ is the set of FIFO and $\mathcal{S}$ is the set of shared memory. Each process $v_i \in \mathcal{V}$ is given by a tuple $v\langle\tau_e,\tau_i,\sigma_c,\mathcal{D}_{sclr},\mathcal{D}_{aggr},\mathcal{D}_{ff},\mathcal{D}_{mem}\rangle$ where $\tau_e(v)$ is actual execution time of the process per packet, $\tau_i(v)$ is idle time of the process per packet, $\sigma_c(v)$ is the total code memory required by the process, $\mathcal{D}_{sclr}$, $\mathcal{D}_{aggr}$, $\mathcal{D}_{ff}$, and $\mathcal{D}_{mem}$ are the set of scalar data items, aggregate data items, FIFO and shared memory, respectively that the process accesses.

$\tau_i(v)$ denotes the idle time spent by the process while it was waiting for an external memory access to complete. $\tau_e(v)$ gives the time consumed by the process to perform actual computation and accesses to local data memory. Thus, the total time required by a process to consume one packet is given $\tau(v) = \tau_e(v) + \tau_i(v)$.

It is the idle time of a process that is amortized by multi-threading. Whenever a thread accesses the external memory, it undergoes a context switch and another thread of the same process is executed. Therefore, based on the preliminary data mapping the total number of threads required to amortize the idle time of a process is given by the ratio $\mathcal{K}(v) = ceil(\tau_i(v)/\tau_e(v)) + 1$. Further, the maximum throughput of a particular process based on the initial data mapping is given by $\mathcal{T}(v) = 1/\tau_e(v)$.

Each scalar variable $d_i \in \mathcal{D}(v)$ (where $\mathcal{D}(v) = \mathcal{D}_{sclr}, \mathcal{D}_{aggr}, \mathcal{D}_{ff}$ or $\mathcal{D}_{mem}$) is given by $d_i\langle\delta,\eta,\sigma\rangle$ where $\delta(d_i,v)$ denotes the average size of each access, $\eta(d_i,v)$

Proceedings of the IEEE Computer Society Annual Symposium on VLSI
New Frontiers in VLSI Design
0-7695-2365-X/05 $20.00 © 2005 IEEE

gives the number of times the data element is accessed during the processing of a packet, and $\sigma(d_i, v)$ denotes the amount of memory required to store the data item. $\delta(v)$ and $\sigma(v)$ are equivalent for a scalar data item. The two quantities are different for the $\mathcal{D}_{aggr}, \mathcal{D}_{ff}$ and $\mathcal{D}_{mem}$.

Each FIFO $f_i \in \mathcal{F}$ is represented by an ordered a pair $f_i\langle v_j, v_k\rangle$ where v_j is the writer process and v_k is the reader. Similarly, each shared memory $s_i \in \mathcal{S}$ is denoted by a tuple $s_i\langle \mathcal{W}, \mathcal{R}\rangle$ where $\mathcal{W}$ and $\mathcal{R}$ are set of writer and reader processes, respectively. A particular process could both read and write to a shared memory.

Objective of the methodology : Given a characterized network processing application $\mathcal{N}$ and a target architecture $\mathcal{A}$, the objective of the methodology is to obtain a mapping function $\mathcal{I} : \mathcal{N} \rightarrow \mathcal{A}$ such that the throughput is maximized. The mapping function maps the processes to PE, and the various data items (code, scalar, aggregate, FIFO, shared memory) to the memory structures of the architecture. The mapping function also specifies the number of concurrent threads of each process on the respective PE.

3.2 Process transformations

We define a pair of process transformations that we apply to the application intermediate format. The transformations are applied with the goal of maximizing the overall throughput of the application. Central to this task is the observation that the overall throughput of the application is determined by the throughput of the slowest process.

Merge transformation : The merge transformation works to consolidate high throughput nodes in an attempt to make the throughput of all nodes more uniform. It takes two neighboring high throughput processes, and merges them into a single, lower throughput process. The steps in performing a merge transformation are:

1. Select two processes $v_i, v_j \in \mathcal{V}$ such that:
   - $\langle v_i, v_j\rangle \in \mathcal{F}$, and
   - $\sigma_c(v_i) + \sigma_c(v_j) < \sigma_c(PE)$,
   - $\sigma(\mathcal{D}_{sclr}(v_i)) + \sigma(\mathcal{D}_{sclr}(v_j)) < \sigma_d(PE)$, and
   - $\tau_e(v_i) + \tau_e(v_j) = min_{\forall(u_i,u_j)\in\mathcal{F}}(\tau_e(u_i) + \tau_e(u_j))$.

   The first conditions ensures that the processes are adjacent, the second condition verifies that the code memory constraint on the PE is not violated, the third condition ensures that the data memory constraint on the PE is satisfied, and finally, the fourth condition selects two processes whose sum of execution time is minimum over all adjacent processes in $\mathcal{V}$.

2. Merge the two processes to generate a new process $v_{i,j}$ such that
   - $\tau_e(v_{i,j}) = \tau_e(v_i) + \tau_e(v_j)$,
   - $\tau_i(v_{i,j}) = \tau_i(v_i) + \tau_i(v_j)$,
   - $\sigma_c(v_{i,j}) = \sigma_c(v_i) + \sigma_c(v_j)$,
   - $\mathcal{D}(v_{i,j}) = \mathcal{D}(v_i) \bigcup \mathcal{D}(v_j)$,
     for $\mathcal{D} = \mathcal{D}_{sclr}, \mathcal{D}_{aggr}, \mathcal{D}_{ff}$ and $\mathcal{D}_{mem}$.

Replicate Transformation : The replicate transformation aims at improving the performance of a low throughput node. It duplicates the lowest throughput node to create an additional copy of the node in the intermediate format. This effectively doubles the throughput of the process the first time this transformation is applied to a node. Each subsequent replication results in a fractional improvement. Let $\mathcal{O}(v)$ denote the number of copies of a process v in the intermediate format. Then, the effective throughput of the process is given by $\mathcal{T}(v) = \frac{\mathcal{O}(v)}{\tau_e(v)}$. The duplicated node v' has the same characteristics as the original node v.

3.3 Mapping and optimization techniques

The objective of mapping the application on the CMP based multi-threaded architecture is obtained in two stages. The first stage, called the process optimization, maps the processes based on the initial data mapping described in Section 3.1. In the second stage, called the data and interprocess communication (IPC) optimization, the data layout is modified to improve the performance.

3.3.1 Process optimization

Process optimization aims at maximizing the throughput by utilizing the merge and replicate transformations. We aim to generate a representation $\mathcal{N}'$ of the application such that number of processes in $\mathcal{N}'$ is equal to the number of PE in the architecture and thereby generate a one to one mapping between processes and PE. The objective is to generate a representation that satisfies the above property and maximizes the throughput. Based on the number of processes in the original application two cases arise.

Case 1: $|\mathcal{V}| > |\mathcal{P}|$ If there are more processes in the application than the number of PE in the target architecture we cannot obtain a one to one mapping as described above. In this case we successively apply merge transformation until the number of processes is equal to the number of PE.

Case 2: $|\mathcal{V}| \leq |\mathcal{P}|$ In the second case the number of processes is less than or equal to the number of PE in the architecture. We successively apply merge transformations to

Proceedings of the IEEE Computer Society Annual Symposium on VLSI
New Frontiers in VLSI Design
0-7695-2365-X/05 $20.00 © 2005 **IEEE**

obtain different application representations with fewer and fewer nodes. Each representation is saved. Thus if $n = |\mathcal{V}|$, we obtain a set $\mathcal{X}\{\mathcal{N}_0, \mathcal{N}_1, \ldots, \mathcal{N}_n\}$ where $\mathcal{N}_0$ is the initial representation and $\mathcal{N}_n$ is the representation with a single process. Next, for each representation $\mathcal{N}_i \in \mathcal{X}$ we successively apply replicate transformation until the number of processes in the representation are equal to the number of PE in the architecture. Thus, we obtain another set $\mathcal{Y}\{\mathcal{N}_0^{'}, \mathcal{N}_1^{'}, \ldots, \mathcal{N}_n^{'}\}$ where $|\mathcal{V}(\mathcal{N}_i)| = |\mathcal{P}|, \forall \mathcal{N}_i \in \mathcal{Y}$. We select the application representation $\mathcal{N}_i^{'} \in \mathcal{Y}$ that has the maximum throughput over all representations in the set $\mathcal{Y}$. The throughput of a representation is given by the lowest throughput over all processes in the representation. As mentioned earlier the effective throughput of a process is given by by $\mathcal{T}(v) = \frac{\mathcal{O}(v)}{\tau_e(v)}$ where $\mathcal{O}(v)$ denotes the number of copies of a process v in the intermediate format. Henceforth, we will refer to the application representation selected at the end of process optimization as $\mathcal{N}_{select}$.

3.3.2 Data and IPC optimization

Data and IPC optimization maximize the throughput by modifying the initial layout. At the end of the process optimization stage the following three situations may arise: (i) idle time of the lowest throughput process cannot be amortized because the architecture does not support enough threads, (ii) idle time of the lowest throughput process cannot be amortized because creation of more threads causes a data spill, or (iii) idle time of the lowest throughput process is completely amortized. The data and IPC optimizations involve movement of data from slower memories to faster memories or vice versa. We associate a priority with each data item that denotes its suitability for a possible movement. In the following section we first discuss the priority associated with each data item and then discuss the optimizations for the three cases specified above.

Priority of a data item : The priority $\mathcal{H}(d)$ of a data item $d_i \in \mathcal{D}_{sclr}, \mathcal{D}_{aggr}, \mathcal{D}_{ff}$, or $\mathcal{D}_{mem}$ of a process v is given by

$$\mathcal{H}_{s\to f}(d) = \left\{\eta(d) \times \delta(d) \times \left[\frac{1}{\omega(m_s)} - \frac{1}{\omega(m_f)}\right]\right\} \times \frac{1}{\sigma(d)}$$

where $\eta(d)$ is the number of accesses to the data item, $\delta(d)$ is the size of each access, $\omega(m_s)$ is the maximum bandwidth supported by the current memory that the data element is mapped to, $\omega(m_f)$ is the maximum bandwidth supported by the next faster memory, and $\sigma(d)$ is the total amount of memory required for storing the data item. The above formulation assumes that the data is moving from a slower memory to a faster memory. The first term (as delimited by the curly braces) denotes the reduction in idle time as the data is moved from a slower to a faster memory. Alternatively, the formulation can also be utilized to move a data item from a current faster memory (m_f) to the next slower memory (m_s) and is denoted by $\mathcal{H}_{f\to s}()$. The last term denotes the penalty associated with allocating the data item to a memory. In the following discussion we either select a data item with highest or lowest $\mathcal{H}()$ value based on the optimization being performed.

Case I: Not enough threads supported : In this case a process has a particularly high amount of external memory accesses that cannot be amortized with the maximum number of threads supported by the given architecture (specified by $\kappa(c), c \in \mathcal{P}(\mathcal{A})$). That is $\lceil \frac{\tau_i(p_m)}{\tau_e(p_m)} \rceil + 1 > \kappa(c)$ where p_m is the least throughput process in $\mathcal{N}_{select}$. In this case the objective is to reduce the idle time by moving FIFO, aggregate data structures, and shared memories from slower memories to faster memory elements. The optimization selects the data item with the *highest* $\mathcal{H}_{s\to f}()$ among all the process's aggregate data items, FIFO and shared memory. The optimization only selects those data items whose movement will not violate the memory size constraint. Further, the optimization does not move data to the on-chip local data memory of the processor. The optimization changes the mapping of the selected data item to the next faster memory, and reduces the idle time of the process by the requisite amount. If the idle time is not amortized the optimization changes the mapping of the another data item. The optimization continues to iterate until the idle time is amortized or no more moves are possible without violating the memory size constraint.

Case II: Not enough memory available : In this case, the introduction of threads to amortize the idle time of the slowest process ($p_m \in \mathcal{N}_{select}$) causes a spill in the on-chip local data memory of the PE. Based on the initial mapping only scalar data items are mapped to the on-chip local memory of the processor. It is assumed that each scalar data item is local to a particular thread. Therefore, as the number of threads required to amortize the process idle time is given by $\mathcal{K}(p_m)$, the lack of enough local data memory denotes the case $\mathcal{K}(p_m) \times |\mathcal{D}_{sclr}(p_m)| > \sigma_d(c)$, where $|\mathcal{D}_{sclr}(p_m)|$ gives the total scalar memory required by a thread of process p_m and $\sigma_d(c)$ gives the total data memory of the processing element. In this case a scalar data item with the *lowest* $\mathcal{H}_{f\to s}()$ value is selected for the move. The execution and idle time of the process are decreased and increased, respectively, by the requisite amounts. The selection and movement of data items continues until the data spill is eliminated and the requisite number of threads are introduced.

0-7695-2365-X/05 $20.00 © 2005 IEEE

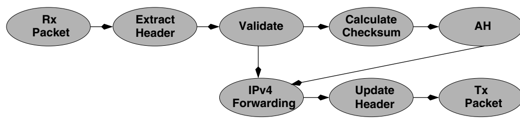

Figure 3. Application I: IPSec and IPv4

Case III: No idle time : In this case we have amortized all the idle time of the slowest process. The only possible mechanism to improve the throughput is by reducing the execution time of the process. This can be achieved by moving the data from local memory to the registers of the process. In this case a scalar data item belonging to $\mathcal{D}_{sclr}(p_m)$ (p_m is the slowest process) is selected for movement to the processing element register. As we assume that each scalar is local to a thread, the optimization assumes that each movement requires $\mathcal{K}(p_m) \times \sigma(d_i)$ number of registers, where $\mathcal{K}(p_m)$ is the number of threads required to amortize the idle time, and $\sigma(d_i$ is the number of registers required by the scalar data item.

4 Results

We demonstrate the effectiveness of our methodology by implementing three network processing applications on the IXP2400 architecture. The applications included i) IPSec authentication and IPv4 forwarding, ii) IPSec authentication, Diffserv conditioning, and IPv4 forwarding, and iii) Diffserv conditioning and IPv4 forwarding. In the three applications IPSec authentication is performed first (except in Application III), followed by Diffserv conditioning (except in Application I), and finally the IPv4 forwarding. The architecture of the IXP2400 processor was discussed in detail in Section 2. Each process in the applications described below was written in micro-engine C. The applications were then profiled in a cycle accurate simulation environment and a process graph was generated for optimization purposes. We illustrate the execution of our methodology by a detailed discussion on the optimization of Application I (IPSec and IPv4). The experimental results for the other two applications are presented after the discussion on Application I.

Application I : The process model of Application I, functionality of each process, and profile information are shown in Figure 3, Table 1, and Table 2, respectively. The number of micro-engines (or processing elements) available on IXP2400 processor is 8. Further, the “Rx Packet” and “Tx Packet” processes cannot be merged with other processes or replicated as they perform special interface functions with the media switch fabric. Therefore, they are not considered for process optimizations. As there are 6 other processes, the process optimization stage considers 6 different designs obtained by merging and replication, respectively. The throughput of the various designs is shown in Table 3 in descending order of the number of processes in the application before replication. As the last design has the highest throughput, it is selected for further optimization. The last design has all processes (except Rx Packet and Tx Packet) merged into one process which is then replicated 6 times. The execution times and idle times of the various processes in the selected design are shown in Table 4. The new process is the merged process that is generated by process optimization stage. As can be noticed from the table, the idle time of the slowest process has been completely amortized. The “Rx Packet” and “Tx Packet” have some idle time which do not effect the final throughput as they are not the slowest processes. The final layout of the major data items in Application I is shown in Table 5. The S7 table, S9 table and expanded key (exp key) are utilized by the F8 algorithm in IPSec.

After the final design was generated by our optimization techniques we implemented it on the IXP2400 architecture. The design was evaluated with a stream of packets using the IXA SDK simulator provided by Intel for 800000 cycles. On implementation, we obtain a measured throughput value of $450Mbps$, which is within 10% of the estimated value of $501.96Mbps$. Therefore, our technique is able to generate performance estimates that are close to the throughput of the final implementation.

Applications II and III: We also applied our methodology to applications II (IPSec, Diffserv and IV4) and III (Diffserv and IPv4), respectively. The final results for all the applications are shown in Table 6. For the second application, we estimated a final throughput of $4818Mbps$, but only obtained a measured throughput of $2700Mbps$. This is due to the maximum bandwidth limitation ($2700Mbps$) of the media switch fabric attached to the network processor. We estimated a final throughput of $469Mbps$ for Application III while the measured throughput was $440Mbps$, which is within 6% of the estimated value. Therefore, our methodology can be utilized to evaluate the performance of different process specifications generated by the designer.

5 Conclusion

As the complexity and functionality of NPs continue to increase, the task of mapping applications onto such platforms becomes increasingly cumbersome. We proposed a comprehensive system-level methodology for mapping applications onto CMP based multi-threaded NP architectures. The methodology considers process level optimizations and addresses the key issues of memory layout, data mapping, and inter-process communication optimization. We implemented three relevant network applications, and

discussed the effectiveness of our methodology. The proposed methodology will enable designers to map network processing applications onto the current NP architectures, without the time consuming process of manually optimizing through experimentation. It will accelerate the design flow for products, leading to shorter design turn around time.

References

[1] Niraj Shah. Understanding network processors. Master's thesis, University of California, Berkeley, September 2001.

[2] Kurt Keutzer Niraj Shah. Network processors: Origin of species. In *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*, October 2002.

[3] C. Matsumoto. Net processors face programming trade-offs. *EE Times, https://www.eetimes.com/story/OEG20020830S0361*, 2002.

[4] Behrooz A. Shirazi, Ali Hurson, and Krishna M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society, 1995.

[5] Giovanni De Micheli, Rolf Ernst, and Wayne Wolf, editors. *Readings in Hardware/Software Co-design*. Morgan-Kaufman, 2001.

[6] Kurt Keutzer Niraj Shah, William Plishker. Np-click: A programming model for the intel ixp1200. In *2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, CA*, February 2003.

[7] Niraj Shah William Plishker, Kaushik Ravindran and Kurt Keutzer. Automated task allocation on single chip, hardware multithreaded, multiprocessor systems. *Workshop on Embedded Parallel Architectures (WEPA-1)*, February 2004.

[8] Intel Corporation. "IXP 2400 Network Processor Datasheet ". *ftp://download.intel.com/design/network/datashts/30116411.pdf*.

Process	Function
Rx Packet	Read packet from MAC device.
Extract Header	Read header information from packet header. For AH packets, includes AH field.
Validate	Verify ethernet address, TTL value, packet type (IPv4 or AH), valid IP source and destination addresses.
Calc. Checksum	Calculate a simple checksum on the IP header.
AH	Decrypt AH. Compare with checksum.
IPv4 Fwd	Look up destination address in IPv4 forwarding table.
Update Header	Remove AH info, decrement TTL, recalculate IP checksum.
Tx Packet	Write packet to MAC device.

Table 1. Application I functionality

Node	Execution Cycles	Idle Cycles
Rx Packet	50	450
Extract Header	15	553
Validate	20	0
Calc. Checksum	50	0
AH	4654	17383
IPv4 Forwarding	107	1033
Update Header	50	0
Tx Packet	60	660

Table 2. Application I: Execution and Idle cycles

Designs	Execution Cycles	Estimated Throughput in Mbps
1	4654	88.01
2	2327	176.02
3	1551	264.08
4	1163	352.19
5	948	432.06
6	816	501.96

Table 3. Application I: Throughput estimates

Process	Execution Cycles	Idle Cycles	Ideal Num. threads	Num. Threads	Idle cycles remaining
Rx Packet	50	450	9	8	50
New Process	4896	18969	3.87	4	0
Tx Packet	60	660	11	8	180

Table 4. Application I: Selected design

Data	S7 Table	S9 Table	IPv4 Table	Exp Key
Location	SRAM	SRAM	SRAM	Local Memory
accesses/packet	192	192	8	64
Size	128	1024	64	2048

Table 5. Data layout for Application I

Application	Estimated Throughput $Mbps$	Obtained Throughput $Mbps$	Variation
IPV4 + AH	501	450	10.1%
IPV4 + Diffserv	4818	2700	Upper limit
IPV4 + AH + Diffserv	472	440	6.7%

Table 6. Final throughput for all the applications

0-7695-2365-X/05 $20.00 © 2005 **IEEE**