Indirect VLIW Memory Allocation for the ManArray Multiprocessor DSP

Nikos P. Pitsianis

Gerald G. Pechanek

Abstract— **The indirect very long instruction word (iVLIW) architecture and its implementation on the BOPS ManArray family of multiprocessor digital signal processors (DSP) provides a scalable alternative to the wide instruction busses usually required in a multiprocessor VLIW DSP. The ManArray processors indirectly access VLIWs from small caches of VLIWs localized in each processing element. With this work, we present an algorithm to perform 1) iVLIW instruction memory allocation on multiple processing elements to minimize instruction memory requirements and 2) scheduling of the iVLIW setup instructions to minimize execution overhead. We present preliminary experimental results that demonstrate the effectiveness of our approach.**

I. Introduction

BOPS system-on-a-chip ManArray cores provide streamlined coprocessor attachment to MIPS, ARM and other host processors. Through selectable parallelism with its indirect VLIW architecture, the ManArray processor cores achieve high performance at low clock rates, which in turn minimizes power requirements. The most efficient performing solution for a given application can be chosen among multiple forms of parallelism, from sequential control code and multiple threads of control, to single instruction multiple data (SIMD) parallelism via packed data types and multiple processing elements, to instruction-level parallelism (ILP) via very long instruction word (VLIW).

A VLIW architecture exploits instruction-level parallelism in a processor. In a VLIW processor multiple execution units can operate in parallel, directly controlled by corresponding fields in the VLIW. Each field may contain a simple instruction to be executed on a specific unit concurrently with the instructions in other fields. The advantage of a VLIW architecture is in the simplicity of the hardware, there is no need for complicated run-time dependency detection as required in a super-scalar architecture. The compiler provides a static schedule of the ILP [1].

There are three major drawbacks associated with traditional VLIW processor architectures. First is the necessity for wide busses connecting instruction memory and execution units. Instruction busses must have the width of the VLIW to transport VLIW contents from instruction memory to the execution units. Consequently, the VLIW bus width can be a significant problem in multiprocessor architectures. Second, special instruction delimiters or no-operations (NOPs) are required that lower the code density and compromise the useful instruction memory bandwidth. Third, VLIWs set a static schedule; although this can be a problem in general purpose microprocessors, it is not an issue for DSPs [2].

The indirect very long instruction word (iVLIW) architecture is a novel way to use the VLIW construct in a distributed SIMD multiprocessor [3]. In general, a VLIW processor requires a wide instruction bus to accommodate the movement of multiple instructions from instruction memory or cache to the execution units. For example, a five-way VLIW, that is a VLIW feeding five execution units, would require $5 \times 32 = 160$ bit wide bus, assuming that processor instructions are 32 bit encodings. On a distributed SIMD multiprocessor, such a wide bus connecting multiple processing elements (PEs) would be very inefficient. Instead in the ManArray, each PE contains a small VLIW cache memory (VIM) so that the 160-bit wide VLIW bus remains local to each PE and is of low capacitance due to short line lengths. A global 32-bit instruction bus connects to each PE, a load VLIW (LV) instruction can load one to five 32-bit instructions into a single VIM line with a 2 to 6 cycle latency and an indirect execute VLIW (XV) instruction causes the appropriate VLIW to be fetched from the local VIMs and executed. A robust instruction set including many application-specific instructions are selectable for inclusion in each distributed VIM providing significant processing efficiency.

In essence, the iVLIW architecture enables a programmer/compiler to prefetch and prepare an instruction cache that is local to the distributed processing elements. Any subset of instructions contained in the same cache line can be executed simultaneously as a VLIW, using only a 32-bit XV instruction with enable bits as a trigger. The iVLIW approach eliminates the necessity for instruction delimiters and NOPs, thus restoring code density and useful instruction memory bandwidth.

The problem of allocating VIM lines and positioning the VLIW preparation instructions to minimize overhead, is a special form of global code motion and resource allocation. We extend the concept of register allocation in a novel way to VIM allocation. Register allocation is the problem of assigning machine registers (hard registers) to program variables (soft registers). Program variables must be loaded onto machine registers because on a RISC processor, execution units can only read and write their operands in registers. Since a processor has a small number of registers, and programs can contain a lot of variables, different variables have to share the same physical register when their lifetimes do not

Nikos@CS.Duke.edu, Department of Computer Science, Duke University, Durham, NC 27708.

LightingHawkCSI@AOL.com, Lighting Hawk Consulting Inc., Cary, NC 27511.

overlap. The VIM allocation problem is analogous to the register allocation problem, VIM is of finite size and program control flow induces VIM dependencies in an similar way to data dependencies. However, unlike register allocation there is the freedom to move VIM initialization statements higher in the program control flow, in a manner similar to data prefetching [4]. We exploit these aspects to develop the VIM allocation algorithm.

We start by a brief introduction to the ManArray architecture in section II and the BOPS iVLIW architecture in section III. In section IV we present an iterative resource allocation algorithm to assign program VLIWs to VIM memory lines and schedule the LV instructions. We conclude with the presentation of preliminary results from the Telecom suite of benchmarks from EEMBC in section V.

II. ManArray Architecture

The ManArray DSP core is a distributed memory multiprocessor with a single instruction stream. One or more PEs operate in lock-step on distributed data using local register files. A special processing element called the Sequence Processor (SP) has program control flow capabilities. Each PE contains a single-cycle, dual-bank data memory and a local VLIW memory. A control and data bus connect the ManArray core with the rest of a System-on-Chip. Figure 1 shows the fundamental building blocks of the ManArray architecture.

Each PE contains five execution units: load (LU), store (SU), arithmetic logic (ALU), multiply-accumulate (MAU) and data select (DSU). The execution units operate in a four or five stage pipeline, where most instructions take either one or two cycles to complete. The operands are 8, 16, 32 or 64-bits wide and are fetched from a local twelve-port data register file (32 registers of 32 bits each, or 16 registers of 64-bits since adjacent even-odd register pairs can be used as 64-bit registers). Address pointers of 32-bits reside in a local eight register address file. Special accumulators operate on 40 and 80 bit fixed point, 32 bit single precision floating point numbers are also supported. The execution units operate on packed-data vectors of eight 8-bit or four 16-bit or two 32-bit or one 64-bit vectors.

Two-level hierarchical conditional (predicated) execution determines whether all or parts of a compute operation on a data vector will be committed, thus reducing the need for program control flow in many occasions. Hardware assisted event-point loops provide high-performance zero overhead loops by maintaining full pipelines and automatic decrement and checking of loop counters for up to two nested loops.

The SP and PE zero coincide and share the execution units, however the SP utilizes separate register files, data memory and instruction memory. The SP can be seen as a PE with the addition of the program counter, the program control flow and interrupt logic.

Instructions are executed in either simplex or VLIW mode. In simplex execution mode, the ManArray core can either execute an SP instruction or a PE instruction (as SIMD). Control flow instructions are executed by the SP in simplex mode only. In VLIW mode, up to five instructions can be executed in parallel on the SP and each PE, while the SP/PE0 can execute combinations of SP and PE instructions that use different execution units.

A distributed cluster switch allows concurrent, single-cycle, register-to-register copy operations among selected pairs of PEs. These structured communication modes include systolic array type of value exchanges along the dimensions of a ring, array, torus, hypercube, binary tree as well as broadcasts. The diameter of the ManArray network is two, i.e. it takes no more than two communication steps for an arbitrary pair of PEs to exchange the values of two registers. The interconnection network of the BOPS multiprocessor DSP is introduced in [5].

A direct memory access (DMA) engine moves data in and out of the PE memories. The DMA has two independent controllers to maintain two separate lanes of 32-bit data flow between all SP and PE local memories and external system memories. The DMA operates in parallel with the ManArray DSP utilizing the cycles that a data memory bank is not being accessed by the DSP. Many address modes are available for flexible gathering and scattering of the data including cyclic, block and bit-reversed address.

The ManArray architecture offers three levels of parallelism (packed-data, VLIW, multiprocessor) that are user-selectable and independent of each-other. Since the ManArray cores are offered as intellectual property, they can be tailored to the exact performance and power requirements of a DSP application. For a more complete presentation of the ManArray architecture, see [6].

The ManArray DSP core is a distributed memory multiprocessor with a shared instruction stream. A long and wide instruction bus would be required in order to accommodate a traditional VLIW architecture, which due to length would not only occupy a substantial chip area but also due to its capacitance, would limit the operational frequency and increase power. The distributed VIM instruction memories alleviate the instruction bus width, but require the penalty of one cycle per instruction to form the required VLIWs. Consequently, once a VIM size has been committed to hardware, it might be insufficient to hold all the VLIWs required by an application. The overhead for reloading the VLIWs can be considered an optimization problem. We designed and implemented a special compiler optimization pass that schedules the VLIW setup program sections to fully utilize the available VIM and minimize machine cycles from reloading VLIWs by maximizing reuse.

III. Indirect VLIW

The iVLIW architecture is implemented in the BOPS ManArray digital signal multiprocessor family. It provides high performance afforded by executing multiple instructions packed in a VLIW and optimizing the implementation for accessing the VLIWs in a multi-PE machine. With iVLIW, the bus width is maintained locally in each PE, at the expense of an additional small VLIW instruction memory near the execution units and the overhead to prepare the contents of VIM prior to execution.

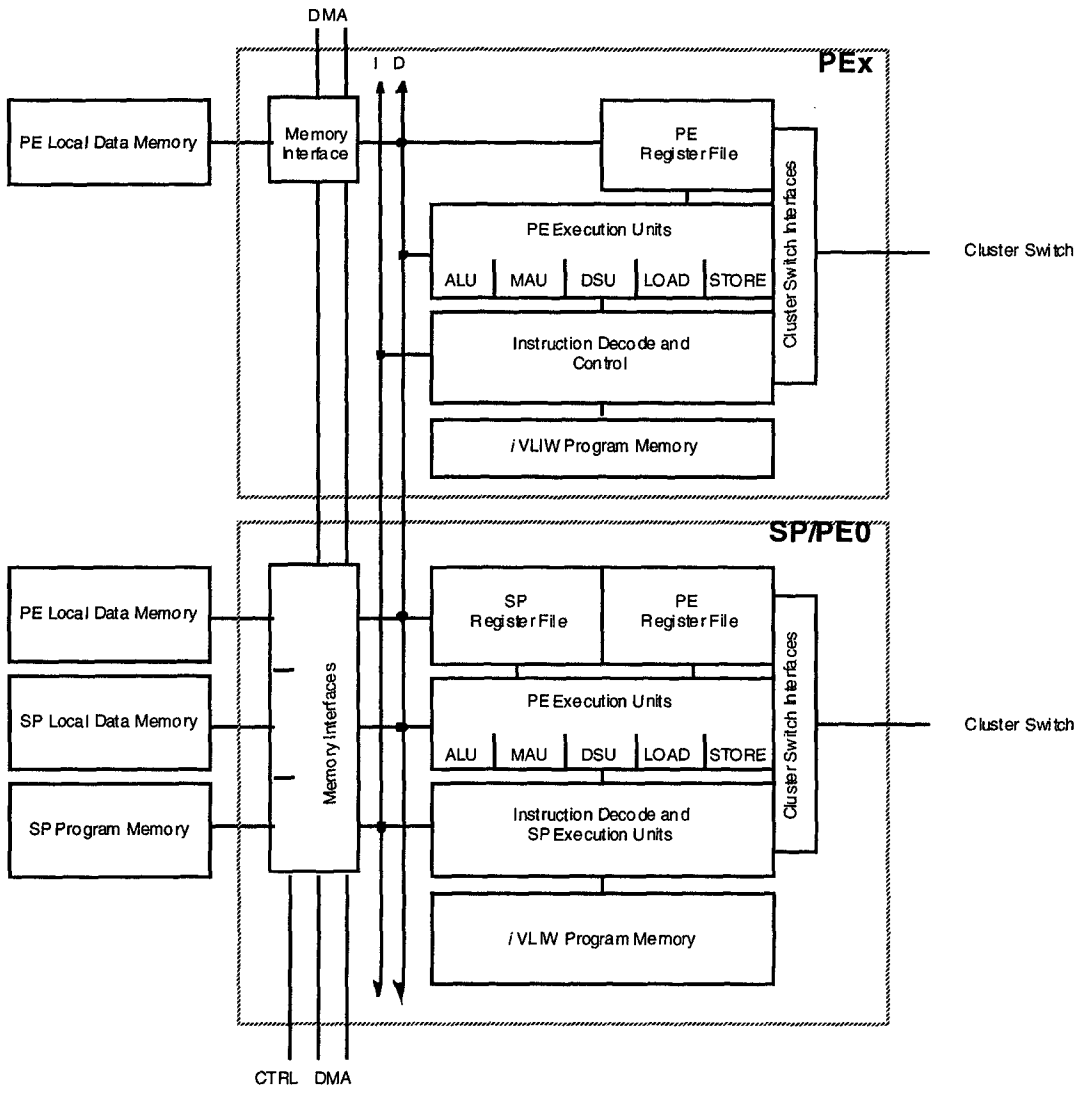

Fig. 1. The ManArray Sequence Processor with Processing Element 0 and a general Processing Element.

Besides providing a scalable VLIW architecture for a multiprocessor DSP, there are other advantages of the iVLIW architecture. Code that uses non-overlapping execution units can be compressed and folded over the same VIM lines. Thus iVLIW not only avoids the explicit storage of non-operational place holders (VLIW delimiters) found in the instruction stream of typical VLIW processors, but also enables code compression by storing more than one non-overlapping VLIW in the same VIM line. Additional code compression is realized by the reuse of the loop body VLIWs in the loop prolog and epilogue via execution-enable flags. Another advantage of the iVLIW architecture in a SIMD multiprocessor system is the fact that one can load different instructions in the same VIM line on different processors. Such a setting permits the synchronous execution of multiple different instructions on different PEs. We have exploited this feature of iVLIW together with register-to-register communications to implement extremely efficient signal transform algorithms such as FFT [7].

More specifically, the BOPS ManArray instruction set contains two special instructions, one to set/modify the contents of a VIM line and another to execute a VIM line. The Load VLIW (LV) instruction is used to load individual instruction slots of the specified sequence processor (SP) or PE VLIW instruction memory address. The VIM address is given as an unsigned 8-bit immediate value offset from a selectable base VIM address register. The number of instructions to load are specified via the instruction count parameter k. For the current generation of the BOPS DSPs, valid values are $1 \leq k \leq 5$. The next k instructions following LV are loaded into the specified VIM in a total of $k+1$ cycles. Additional parameters specify whether an execution unit is enabled and which of the compute

execution units would provide the output flags when this VIM line is executed. The assembly program listing in Figure 2 contains two LVs with mnemonic `lv.p`, the first one prepares PE VIM line 1 with a load immediate indirect and a store immediate indirect memory instructions and the second LV prepares PE VIM line 0 with a load immediate indirect and an add instructions.

The execute VLIW (XV) instruction is used to execute an iVLIW stored at individual addresses of the specified SP or PE VIM. As stated above, the VIM address is computed as the sum of a base VIM address register and an unsigned 8-bit offset immediate value. Additional parameters specify any combination of individual instruction slots to be executed as well as a parameter to override the unit affecting the flags to be returned by the execution of the XV. The default unit affecting the flags to be returned is set by the corresponding LV that last modified the VIM line executed by the XV instruction.

The assembly program in Figure 2 implements the addition of two independent vectors of length 256 containing 16-bit integers onto a third vector. The result is attained with a hardware-assisted zero-overhead loop consisting of two VLIWs labeled `_BEC_2` and `_BEC_1`. Observe that the same two VLIWs are utilized repeatedly. In the loop prologue, only the load instructions of the VLIWs are executed using the enable flag `e=l` of the XV instruction. In the loop body, all instructions are executed and in the loop epilogue, only the final add (`e=a`) and store (`e=s`) instructions of the two VLIWs are executed.

To assist the reader understand the assembly code of this example we note that `lim` denotes a load immediate; instructions `eploopi0` and `sspr` set up the loop counter value, end address and beginning address of the hardware-assisted zero overhead loop respectively; `lii` and `sii` are load and store instructions with automatic pointer update; and `add` is the addition. The `e=` flags of the `xv` instructions enable the execution units `l` for the LU, `s` for the SU, `a` for the ALU. Infixes `.s` and `.p` denote that the instruction is executed by the SP or the PEs respectively. Postfixes `.w` and `.h` denote the data type of the operation. The arguments of the function are passed on pointers `A2` for the target vector and `A0` and `A1` for the two input vectors.

For simplicity, we did not use packed data in this example. With packed data, four half-words can be operated on simultaneously in each PE.

```
        .section .text
__vadd:
        lim.s.w                 R0, _BEC_2
        copy.pd.w               A3, A0
        copy.pd.w               A0, A2
        eploopi0                63, _BEC_1
        sspr.s.w                R0, IEP0R1

VLIW1: lv.p V0, 1, 2, d=, f=n
               sii.p.h0         R1, A2+, 1
               lii.p.h0         R1, A1+, 1
VLIW0: lv.p V0, 0, 2, d=, f=n
               lii.p.h0         R0, A0+, 1
               add.pa.2h        R1, R1, R0

        copy.pd.w               A2, A3
prolog:
        xv.p V0, 0, e=l, f=
        xv.p V0, 1, e=l, f=

        !!!! Loop Begins
_BEC_2:
        xv.p V0, 0, e=la, f=
_BEC_1:
        xv.p V0, 1, e=ls, f=
        !!!! Loop Ends
epilogue:
        xv.p V0, 0, e=a, f=
        xv.p V0, 1, e=s, f=

        ret
```

Fig. 2. A leaf function with two load and execute iVLIW statements.

IV. VIM Allocation and Code Motion

In Figure 2 assembly function `vadd` sets up and uses two iVLIWs. When function `vadd` is executed many times, it would be beneficial to relocate the LV instructions to program sections that call `vadd` causing them to be executed fewer times. As a result, the total execution time of a program using function `vadd` is reduced. The VIM setup is pushed higher in the program control flow graph (CFG) so that program sections that execute frequently do not have to re-initialize their VIM. However, in a typical hardware implementation the size of VIM is limited, just like the number of registers and the number of execution units. One may not be able to move all VIM initializations at an early set-up phase of a program execution, but rather allocate VIM use and distribute its initializations in a manner similar to register allocation.

The need for VIM allocation and relocation of LV statements arises in all back-end compiler tools that generate VLIW code and hand-written assembly code. Code generators in compilers and assembly programmers in large software projects, usually have a local or limited view of the whole program. The tool we are presenting here applies inter-procedural analysis, and it can be used with any compiler that generates VLIW code for the ManArray.

The *control flow graph* is a directed graph $G(N, E)$ where node N is a function or program basic block and edge $e(n_1, n_2)$ from E is a jump or a function call from basic block n_1 to n_2. The graph contains a special program-start node (with no parents) and an end node (with no descendants). A *basic block* is an indivisible sequence of program instructions with a single entry, the beginning of the block and a single exit at the end of the block.

We approach the problem of VIM allocation as follows: First we construct the program CFG. Then we find the VLIW live-in and the live-out sets for each node of the CFG, we call the augmented graph as the *VLIW flow graph*. From the VLIW flow graph we construct the VLIW interference graph which we color to derive the VLIW allocation to VIM. The LV instruction scheduling requires the iterative recalculation of the VLIW flow graph and subsequent allocation.

The *life of a VLIW* is defined to be the interval between the time a VLIW is defined via an LV instruction, until the time it is executed for the last time with an XV instruction.

The liveness analysis is performed by solving the VLIW flow equations [8]. The VLIW flow graph captures the dependencies between the VLIW instructions similar to the data flow graph. The presence of an LV instruction defines a VLIW in a node while the presence of an XV instruction uses a VLIW in a node. We will refer to these as a def-use pair. For every node n of the CFG we define

- D_n the set of VLIWs that are defined in node n,
- U_n the set of VLIWs that are used in node n,
- I_n the set of live-in VLIWs to node n, these are the VLIWs that are defined in a predecessor to node n – with respect to the CFG– and are either used in node n or in a successor node and
- O_n the set of live-out VLIWs to node n, the VLIWs used in a successor to node n.

The sets I and O are the fixed points of the *VLIW flow equations:*

$$I_n = U_n \cup (O_n - D_n)$$
$$O_n = \cup_{s \in S(n)} I_s.$$

The VLIW flow equations state that the set I_n of live-in VLIWs at node n, is the union of the sets U_n of the VLIWs that are used in n and O_n the VLIWs that live-out of n except D_n the VLIWs that are defined in node n. The set O_n of live-out VLIWs of node n are all the VLIWs that belong to the live-in sets of the successor nodes of n, denoted as $S(n)$. We solve the VLIW flow equations iteratively with starting values $I_n = \emptyset$ and $O_n = D_n$ for all nodes n.

The VLIWs that belong to the live-out set of a node, cannot be assigned to occupy the same VIM line unless they do not require the same execution units. To determine how to allocate the VIM lines, we build the *interference graph* [9]. Every node of the interference graph corresponds to a VLIW. When two VLIWs belong to the live-out set of the same node of the VLIW flow graph and share the same execution unit, an undirected edge is inserted in the interference graph connecting the corresponding VLIW nodes. A successful allocation of VIM lines to VLIWs corresponds to coloring the interference graph nodes in such a way that adjacent interference nodes are colored in different colors. Each color corresponds to a different VIM line.

An optimal placement of VIM initialization instructions with respect to given profiling data involves calculating the total number of times each LV instruction is executed and moving it to a program region where it is executed fewer times. We accomplish this via a greedy optimization algorithm that, at each iteration moves the LV instruction that has the highest count of executions to a basic block that is higher in the CFG with a smaller count of executions. Such a move modifies the interference graph, and a new VLIW allocation is required. The algorithm in Figure 3 iterates until no further improvement can be found. At every iteration, an LV instruction is chosen to be moved. The criterion for the selection of the LV is the frequency of executions from run-time profiling data. In the absence of profiling data, we use heuristics, the distance of a node from a leaf node and the depth of nested cycles that a node or edge belongs to. The LV is placed in a basic block that is closer to the program start node and the following conditions are met: i) the execution frequency of the new node is lower and ii) the VIM allocation succeeds in assigning a VIM line without exceeding the maximum number of VIM lines. The algorithm terminates because either all LV statements have reached the program start basic block, or any movement of an LV statement to a node with lower frequency results to an interference graph that cannot be colored with the available number of colors.

```
done := false;
while not done do {
  done := true;
  BestImprovement := 0;
  for each LVi from LVlist do {
    [NewState, improvement] := MoveUp(LVi, CurrentState);
    if improvement > BestImprovement then {
      BestState := NewState;
      BestImprovement := improvement;
      done := false;
    }
  }
  if not done then {
    CurrentState := BestState;
  }
}
```

Fig. 3. Pseudo-code for steepest descent search algorithm for iVLIW allocation and LV scheduling.

The iterative optimization is steepest hill descent and does not guarantee convergence to the global optimum. Also, since graph coloring is required, the worst time running complexity of this algorithm is exponential to the number of VLIWs. Suboptimal graph coloring reduces the run-time complexity of the algorithm to polynomial time in the number of VLIWs and the number of the nodes of the CFG. We instead implemented a problem size reduction by graph compression of "uninteresting" nodes of the CFG. A node is considered *interesting* if it contains LVs or XVs or calls to functions with interesting nodes. We coalesce adjacent uninteresting nodes to reduce the problem size.

The running time complexity of VIMA has not been an issue because a VIM optimization pass is required at the final stages of a project completion, well after other major computation and performance issues have been resolved.

V. Results

We implemented the VIM allocation and LV scheduling algorithm as a stand-alone optimization tool we named VIMA, for VIM Allocator. VIMA is invoked just before the linking phase of a project. Although VIMA can be part of a whole-program optimizing compiler, we kept it as a separate tool to process code written in multiple programming languages as well as assembly. Product-quality DSP software often requires several modules to be coded in assembly language.

We tested the performance improvement due to VIM allocation and LV relocation compared to LV statements scheduled prior to their corresponding XV instructions. The VIM size was constrained to 64 lines in both the SP and each of the PEs.

TABLE I

MACHINE CYCLES OF RUNTIME AND PERCENTAGE OF PERFORMANCE IMPROVEMENT OF THE DIFFERENT DATASETS OF THE EEMBC TELECOM BENCHMARK, USING VIMA TO SETUP VLIW INSTRUCTION MEMORY.

Benchmark	VIMA	w/o VIMA	Change
AutoCorr	56	68	17.65%
	2090	2110	0.95%
	2068	2115	2.22%
ConEnc	46	57	19.30%
	41	50	18.00%
	37	43	13.95%
FFT	683	797	14.30%
Viterbi	4594	4622	0.61%
FBitAl	15600	16545	5.71%
	3706	4606	19.54%
	14113	15733	10.30%

We used the Telecom set of standard industry benchmarks in C by the Embedded Microprocessor Benchmark Consortium (EEMBC). EEMBC develops and certifies real-world benchmarks and benchmark scores to help designers select the right embedded processors for their systems [10]. The Telecom is one of several benchmark suites that are targeting a specific market segment. Other suites are Automotive/Industrial, Consumer, Networking and Office Automation. The Telecom suite consists of the following kernels: autocorrelation, bit allocation, fast Fourier transform, Viterbi decoder and a convolutional encoder. The benchmark codes were compiled with the BOPS Halo C compiler. Table I lists the machine cycles of runtime and percentage of performance improvement of the different datasets of the EEMBC Telecom benchmark kernels when VIMA is utilized. Different datasets exercise different parts of the benchmark code. Although the set of benchmarks is rather small, an embedded DSP in a Telecom application is spending a great percent of its productive cycles in program kernels like these [11]. We did not use profiling to determine the execution frequencies of the different program regions.

The comparison of the VIMA performance improvement on the compiled C kernels is against code where the VLIW setup LV statements are scheduled in the basic block containing the prologue of the pipelined loop where the corresponding VLIW is used. These basic blocks may themselves be inside outer loops.

We expect performance increases and even greater dependence on iVLIW scheduling when the BOPS Halo C compiler generates VLIW instructions for acyclic code; the version of the compiler we used in these tests exploits ILP for explicit `for` and `while` loops only.

For a significantly larger code test, we used VIMA on the hand-coded assembly source of the G.729A voice codec [12]. We observed similar performance gains to the EEMBC benchmarks in the order of $15-20\%$ for different test vectors. The LV overhead penalty was minimal, despite the fact that the implementation of G.729A used requires 300 VLIWs, a large multiple of VLIWs than the 64 line size of the available VIM. The performance comparison for the assembly coded G.729A is against the manual scheduling of the LV statements within the procedure modules where they appear. The performance improvement was due to the whole-program inter-procedural analysis of VIMA as the different programming teams have accomplished efficient intra-procedural VIM allocation.

All experiments were performed on both a cycle-exact software simulator and a Xilinx FPGA as well as the first silicon of the ManArray architecture called Manta. The Manta DSP consists of 4 PEs and operates at 133MHz, it is built by TSMC using standard $0.25\mu m$ libraries. The Telecom benchmark kernel results are independently verified and certified on Manta by the EEMBC benchmark consortium.

VI. CONCLUSIONS

The iVLIW architecture permits the efficient use of the VLIW concept in a high-performance and low-power multiprocessor DSP. The use of VIM eliminates the need for wide instruction busses that cross the multiprocessor and alleviates the instruction memory bandwidth penalty for storing and fetching the required delimiters or NOPs of ordinary VLIWs. The associated overhead for preparing the VIM contents can be all-but eliminated with the use of straightforward allocation and scheduling, as these preliminary results indicate.

VII. ACKNOWLEDGMENTS

The authors wish to thank the entire technical staff of BOPS Inc and the Duke University student interns Sanjay Banerjee and Benjamin Strautin for their contributions to this project.

REFERENCES

[1] J. A. Fisher, "Very long instruction word architectures and the ELI-512," in *Tenth International Symposium on Computer Architecture*. Computer Society Press, 1983, pp. 140–150.

[2] Jeff Bier, ""VLIW Architectures for DSP: A Two-Part Lecture Outline"," in *International Conference on Signal Processing Applications and Techniques*, Orlando, FL, 1999.

[3] Edwin F. Barry and Gerald G. Pechanek, "Methods and apparatus for instruction addressing in indirect VLIW processors," US Patent 6,356,994, March 12 2002.

[4] D. Callahan, K. Kennedy, and A. Porterfi eld, "Software prefetching," in *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 40–52.

[5] Gerald G. Pechanek, Stamatis Vassiliadis, and Nikos P. Pitsianis, "ManArray interconnection network: An introduction," in *EuroPar '99 Parallel Processing*, Aug. 31-Sept. 3 1999, vol. 1685 of *Lecture Notes In Computer Science*, pp. 761–765.

[6] Gerald G. Pechanek and Stamatis Vassiliadis, "The ManArray Embedded Processor Architecture," in *Proceedings of the 26-th Euromicro Conference: "Informatics: inventing the future"*, Maastricht, The Netherlands, September 5-7 2000, vol. I, pp. 348–355.

[7] Nikos P. Pitsianis and Gerald G. Pechanek, "High-performance FFT implementation on the BOPS ManArray parallel DSP," in *Advanced Signal Processing Algorithms, Architectures and Implementations IX, SPIE International Symposium*, July 1999, vol. 3807, pp. 164–171.

[8] Andrew W. Appel, *Modern Compiler Implementation in ML*, Cambridge University Press, 1997.

[9] G. J. Chaitin, "Register allocation and spilling via graph coloring," in *ACM SIGPLAN 82 Symposium on Compiler Construction*. ACM, June 1982, pp. 98–105.

[10] Alan R. Weiss, "The standardization of embedded benchmarking: Pitfalls and opportunities," in *Proc. IEEE Int. Conference on Computer Design (ICCD'99)*, 1999, pp. 492–498.

[11] Markus Levy, "Processor benchmarks for key embedded applications," *Embedded Systems Design*, pp. 55–56, May 2001.

[12] "ITU-T Recommendation G.729, CS-ACELPD," March 1996.