Utilizing Memory Bandwidth in DSP Embedded Processors

Catherine H. Gebotys
Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario Canada
cgebotys@optimal.vlsi.uwaterloo.ca

ABSTRACT

This paper presents a network flow approach to solving the register binding and allocation problem for multiword memory access DSP processors. In recently announced DSP processors, such as Star*core, sixteen bit instructions which simultaneously access four words from memory are supported. A polynomial-time network flow methodology is used to allocate multiword accesses while minimizing code size. Results show that improvements of up to 87% in terms of memory bandwidth (and up to 30% reduction in energy dissipation) are obtained compared to compiler-generated DSP code. This research is important for industry since this value-added technique can increase memory bandwidths and minimize code size without increasing cost.

1. INTRODUCTION

Due to increasing complexities of domain applications, high level language compilation is a necessity for DSP processor cores. However the biggest drawback to DSP processor cores is the lack of efficient optimizing compilers. The use of conventional code generation techniques and even compilers specifically designed for commercial DSP processors are known to produce very inefficient code[1].

The code generation problem has tight constraints due to DSP architectural features as well as price, performance, and power requirements. In more recently announced DSP processors, constraints placed upon code generation include dual bank register files, higher memory bandwidths available for aligned sequential data words, and execution set overheads. DSP processors have steadily been increasing the number of functional units in their architecture to support higher levels of parallelism. However only recently has a somewhat equivalent increase in memory bandwidth been available. For example a DSP processor[5], which has four complex functional units, can fetch up to 8 memory aligned words in a single cycle (thus supporting access of two operands per functional unit per cycle).

Multiword memory accesses provide high memory bandwidth matching computational throughputs of recent embedded DSP processors. However due to compact code size requirements, this multiword capability places additional constraints on data register allocation and binding. For example in the Star*core processor[5] (jointly developed by Motorola and Lucent), 16 bit quad memory access instructions are supported. These quad memory access instructions load or store 4 memory aligned (16 bit) data words with only one instruction. Dual memory access instructions require that the first of two data words (aligned in memory) must be loaded into an even numbered data register (such as *d2*) and the second of two data words must be loaded into the adjacent odd numbered data register (such as *d3*, adjacent to *d2*). Alternatively a dual load could be used with data register pairs (*d0*,*d1*) or pair *(d4,d5)*, etc. In the quad memory access instruction the first and third of four data words (aligned in memory) must be loaded into even numbered data registers and the second and fourth of four data words must be loaded into the adjacent odd numbered data registers. Each quad memory access has a choice of only four sets of data registers it can load into (or from if a store is being performed) out of a total of 16 data registers available in the register file[5]. These sets are registers *(d0,d1,d2,d3)* or set *(d4,d5,d6,d7)* or *(d8,d9,d10,d11)* or *(d12,d13,d14,d15).* If the compiler cannot bind the data to one of these four sets, obeying both memory layout and alignment restrictions, then it must use several single or dual memory access instructions. Since each memory access instruction whether quad or single is 16bits, code size can be reduced through efficiently utilizing dual and quad memory access instructions where ever possible.

In summary the multiword memory access instructions have an important impact on code size which in turn effects cost, energy and power dissipation. The need for decreasing time to market, development costs, and maintenance costs, demands the use of efficient high level language compilation for these sophisticated DSP processors which support these multi-word access constraints. All of these factors imply several challenges in writing efficient code generators for such DSP processors.

2. PROBLEM AND RELATED WORK

The following problem, problem 1 given below, is an important part of the code optimization problem for DSP processors that will be studied in this paper. For the problem definition below we assume that there is one target DSP processor or core defined with an instruction set architecture, such as the Star*core processor[5]. The target processor supports dual and quad memory accesses (of aligned and sequential 16 bit words) with aligned data register constraints. In the general application, it

Conference '00, Month 1-2, 2000, City, State.

is assumed that some data has a predetermined memory layout (for example input data or output data of a DSP algorithm), with a fixed or flexible alignment. Other data may have no predetermined memory layout or alignment, such as data coefficients which can be stored in memory according to the designer, compiler or CAD tool.

Problem 1 : Assume we are given initial assembly code generated for the target processor, including memory layout (and alignment) of the data. The problem is to improve the register allocation and binding code such that the code size is minimum. Specifically allocate and bind registers, with constraints of both fixed and flexible data memory layout, to maximize the number of quad and dual memory access instructions such that the total resultant code size is minimized. Extensions to this problem include handling loops, conditional code, and data structures.

Although many researchers have studied code generation for DSP processors or ASIPs [1], fewer have studied memory-access code generation. A number of researchers have examined allocation of address registers[7,12], data layout [4] , and index registers[9] . Researchers in [11] introduced a transformation of C code using pointer based code instead of array based code to make better use of address calculation units on DSPs. Register binding and allocation for low energy was studied in [10], where minimization of the bit switching between pairs of variables which shared the same register was performed. A network flow approach was used, however memory accesses were not considered. Researchers in [12] applied a minimum cost circulation technique to optimize address register allocation. Results showed significant improvements in performance, code size and energy dissipation. Register binding and instruction scheduling is researched in [2] to minimize the number of registers and find tight schedules through the addition of sequencing relations. Memory bandwidth minimization for video and communications applications is studied in [3], in order to reduce the cost and complexity of memory architectures.

In this manuscript a new problem, memory-access register allocation and binding, is defined and solved. Unlike previous research, we study the problem of given flexible and fixed data layout in memory, generate optimal memory-access code to minimize code size. A maximum cost flow technique is used to obtain solutions in polynomial time. The next section will outline the assumptions and terminology to be used in the rest of the paper.

3. TERMINOLOGY AND ASSUMPTIONS

The following terminology will be used in this paper: Variables, *i*, can be data stored in registers (produced by the result of an instruction, accessed by other instructions, modified by multiply-accumulate type instructions, values moved to/from memory or coefficients used in the code). A variable, *i*, can have a number of attributes such as: a lifetime which is a range of time from *define_time(i)* to *last_used_time(i)* where *define_time(i)* is the cycle in which the variable is defined and *last_used_time(i)* is the cycle in which the variable is last used ; *access_times(i)* are the set of cycles during which the variable is accessed or used. Additional terminology to be used in the optimization discussions are: *G=(V,A)* is a graph *G* composed of vertices *V* and arcs *A*. The variable $x_{i\rightarrow j}$ is the flow in the arc $i\rightarrow j$ from vertex *i* to vertex *j*, where $i,j \in V$ and $i\rightarrow j \in A$. The value $c_{i\rightarrow j}$ is the capacity on arc $i\rightarrow j$. The value $e_{i\rightarrow j}$ is the cost per unit of flow on arc $i\rightarrow j$. There are two special vertices in this graph called vertex *s* and vertex *t*. The flow out of vertex *s* is equal to the flow into vertex *t* and for all other vertices the flow in is equal to the flow out. Arcs incident to *s* only leave vertex *s* and arcs incident to vertex *t* only are directed into vertex *t*.

The maximum cost flow problem[15,16] is to fix the amount of flow through the graph and maximize the sum over all arcs of the flow multiplied by the cost. As long as the capacities, $c_{i\rightarrow j}$, and the lower bounds on the flow, ($l_{i\rightarrow j}$), are integer, we can be guaranteed of obtaining integer flows in the solution of this problem[6]. This problem can be solved in polynomial time using linear programming or network algorithms. Applications of network flow have been used to solve a large number of difficult problems[14].

4. MODELING AND METHODOLOGY

This section will briefly describe the methodology for DSP memory-access code optimization, problem 1, and how the maximum cost flow formulation is used to solve problem 1. The methodology first determines a lower bound on the number of registers, $|R|$, required in the application code (based upon variables lifetimes, using network flow). It proceeds to use this number as a fixed amount of flow and maximizes a cost equivalent to the number of dual memory access instructions which can be supported by solving a maximum cost network flow problem. It next proceeds with the same fixed amount of flow and maximizes a cost equivalent to the number of quad memory access instructions in the application. In the final stage of the methodology, with the dual and quad memory accesses fixed (from the solutions of the two previous max-cost flow problems), the register binding is performed by solving several network flow problems with fixed flows ranging from *1* to *|R|-2(#dual)-4(#quad)* and costs representing the sum of the number of accesses of each variable in the binding. This final cost is attempting to minimize the number of register accesses of higher numbered data registers, since accesses to registers in the upper bank have a prefix overhead[5] (or extra 16 bit word required which increases the code size). Thus we use the cost formulation to allocate more frequently accessed variables into the lower bank of registers.

To illustrate how problem 1 can be modeled as a flow problem, we first have to develop a network flow graph. Each vertex, $v_i \in V$, in the graph corresponds to a variable, *i*, in the assembly code. Vertices *s* and *t* are added to the graph representing times *0* and *s+1* , where *s* is the number of execution sets in the application assembly code, respectively. Arcs from the *s* vertex to all vertices in the graph (except vertex *t*) are added. Arcs are formed from each vertex v_i to all other vertices v_j such that *last_used_time(i)= define_time(j)* (or in other words variables *i* and *j* can share the same register since their lifetimes do not overlap). The capacities of all arcs are set to *1*. In the min-cost solution, each path of flow through vertices represents one register which stores the respective variables. For illustration purposes the data registers are represented by *d0* through *d7*. Single memory access instructions (load or store) will be

represented by *move.f*. Dual and quad memory access instructions will be represented by *move.2f* and *move.4f* respectively.

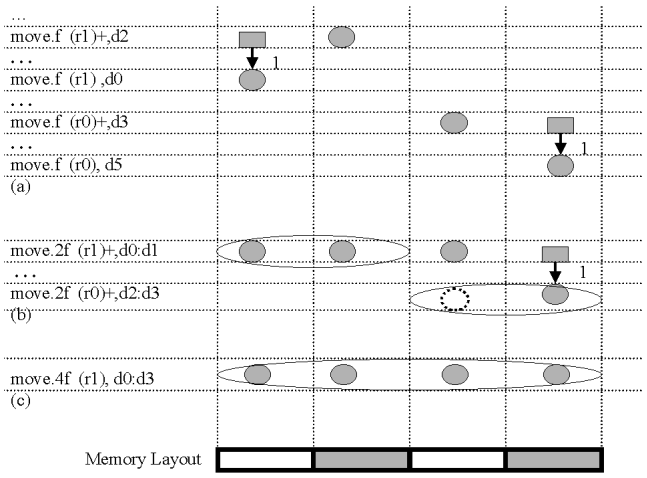

Figure 1. Network flow with dual costs in (a), with quad costs in (b) and finally quad solution in (c) for fixed memory layout.

Variable lifetimes are extracted from the compiler generated assembly code to create the network flow graph. Additional vertices are added into this graph in order to increase the number of multiword memory accesses. The maximization of multiword memory accesses will be briefly described. The horizontal axis in figure 1 represents the memory layout. Specifically increasing (byte addressable) memory address locations are shown as boxes from left to right. The first box on the left represents an even memory address in binary whose last three bits end in "000", which we will represent as –000. The next three adjacent boxes represents memory addresses -010, –100 and –110 respectively. The vertical axis of figure 1a), 1b) and 1c) each represents part of the assembly code increasing in clock cycles downwards. In figure 1 the column on the left represents part of an assembly code program, only showing the loads (ie *move.f (r1)+,d2* , where *r1* is address register and *d2* is data register). The circles represent a vertex in our network flow graph which is a variable being loaded from memory (at the address identified on it's *x-axis*, with the code located on the *y-axis*) in this example. Arcs into and out of this circle are not shown for simplicity. The squares shown in figure 1 represent the additional vertices placed in the network flow graph. In figure 1 the square vertex represents an earlier load of a variable, which is represented by the circle located vertically below it and connected to it by the arc. In the network flow solution, a flow through these squares represents a multiword memory access instruction (since the earlier memory accesses can be performed in parallel with the other memory access). Thus a cost of one is assigned to this arc (connected from the square to the circle). In the network flow problem we maximize the number of multiword accesses which is the sum of costs on these arcs multiplied by the flow on these arcs.

For example a flow through this arc transforms the two *move.f* (single word memory access) instructions, in figure 1 (a), into one *move.2f* (dual memory access) instruction. The *move.2f's* are represented in figure 1b) as oval circles, each containing two circles. The total cost is the savings in code size words (ie. the total number of *move.f* words saved, since 2 are replaced by one *move.2f* instruction) or equivalently the number of dual memory access instructions allocated. In the next step we take two *move.2f* instructions and add one arc with a cost of one to create one *move.4f* or quad memory access instruction, as shown in figure 1b). Again the objective of this second network flow problem is to maximize the number of quad memory access instructions allocated. Note in figure 1b) if the flow in the arc of cost 1 is zero, then we still have a valid allocation of memory access instructions since we would use two dual instructions (illustrated by the dashed circle). However even when the dual and quad memory accesses are maximized using the technique described above, the final code cannot be generated until register binding is performed (which specifically assigns a data register from the bank to each variable), which will be described next.

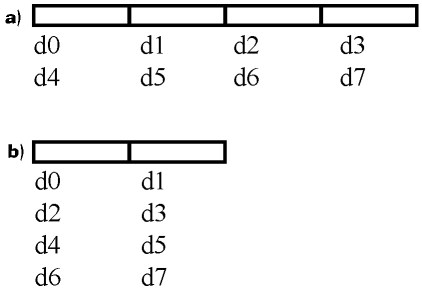

Figure 2. Register assignments in quad memory access instructions (a), and dual memory instructions (b).

After the network flow has been used to allocate registers and dual/quad memory accesses, the register binding problem must be solved. In figure 2 the possible register bindings for each dual and quad memory access field are shown. For the register binding problem, we also use a network flow formulation. Each flow through our graph is a binding of a register *di* to those variables represented as vertices the flow goes through. The methodology incrementally binds one or more registers at a time. We consider even and odd flows which specifically are binding of even (*d0,d2,...d6*) or odd (*d1,d3,...d7*) numbered registers respectively. In figure 3 a) one quad memory access (shown by four boxes in a row at the top, representing the four memory layout locations, with odd numbered registers shaded) and three dual memory accesses are shown. In the network flow graph, vertices for all the other data variables are a part of the flow graph, however will not be shown in the figures for simplicity. In figure 3a) a total flow going in from the top vertex (*s* shown as a circle) and out of the bottom vertex (*t* shown as a circle) is 2, which we call the even flow since it represents flow through two even data registers (*d0,d2*). A maximum cost flow problem is solved where the number of accesses on each variable is the cost of that vertex, being maximized. In figure 3b) the flows of the solution are shown. In this case we have allocated data registers *d0,d2* as shown. Next we must bind data register *d1* and *d3*. This next problem we solve two network flow problems, the first with single flow through *d1* variables of dual or quad memory accesses as shown in figure 3c) and the second for *d3* shown in figure 3 d).

Consider two quad memory accesses, where the set of variables in one quad memory access do not have lifetimes which overlap with any variable in the second quad memory access variable set. We will use the term non-overlapped quad memory accesses for this case. Furthermore if these two quad memory accesses have fixed memory layouts then a single flow formulation must be used, as

shown in figure 4a). In figure 4 b) the quad memory accesses overlap, thus flows of 4 can be used through the even variables. After all dual and quad memory access variables are bound to registers the remaining variables are allocated using network flow.

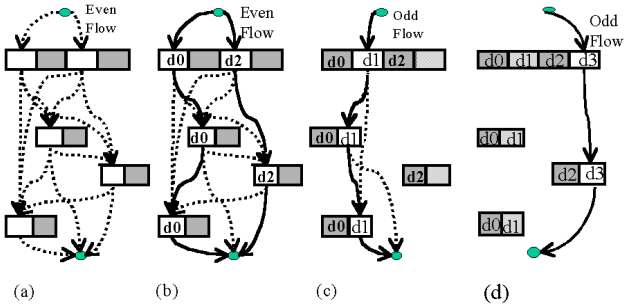

Figure 3. Even flows in a) binding registers d0,d2 in (b) and odd flow in c),d) binding registers d1,d3 respectively.

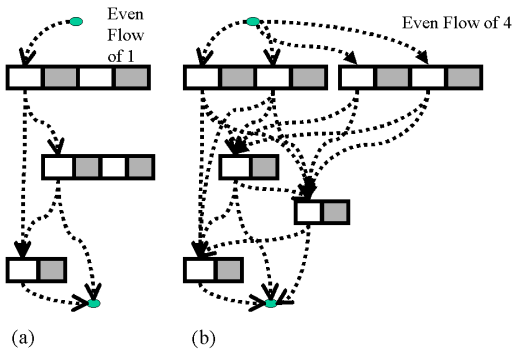

Figure 4. Non-overlapping quad accesses requiring single flows in (a), but in (b) overlapping quad accesses use flow of 4.

We can now formulate the register binding network flow problem. We define the set of variables which have been given the possibility of flow as set F, where variable $k \in F$. If a variable cannot be bound or allocated in the specific network flow problem, we use the terminology, $k \notin F$. We define our costs, $e_{i \to j}$, as the number of accesses of that variable j (as explained earlier). For example if we are solving the problem in figure 4b), we define variables, $k \in F$, as all even variables of quad or dual memory access instructions with fixed memory layouts, plus all variables of dual or quad memory access instructions with flexible memory layouts, plus all other variables in the graph extracted from variables of the assembly code. The set of variables with no flow or $k \notin F$ are those which are the odd numbered variables in dual/quad memory access instructions with fixed memory layouts. We denote a variable, k , which is a member of a quad or dual access instruction, i, as $k \in Q_i$, where Q_i represents the set of variables in multiword memory access instruction i. Finally R is the number of registers we are currently binding (for example in problem in figure 4a) $R=1$ whereas in figure 4b) $R=4$.

$$\text{Maximize } \Sigma_{i \to j} \; e_{i \to j} \; x_{i \to j}$$

$$\Sigma_{i|i \to j} \; x_{i \to j} - \Sigma_{i|j \to i} \; x_{i \to j} = 0, \; \forall j \in V$$

$$\Sigma_{i|i \to v_k} \; x_{i \to v_k} \leq 1, \; \Sigma_{i|v_k \to i} \; x_{v_k \to i} \leq 1, \; \forall v_k \neq s,t, \; v_k \in V, \; k \in F$$

$$\Sigma_{i|i \to v_k} \; x_{i \to v_k} \leq 0, \; \Sigma_{i|v_k \to i} \; x_{v_k \to i} \leq 0, \; \forall v_k \neq s,t, \; v_k \in V, \; k \notin F$$

$$0 \leq x_{i \to j} \leq 1, \; \forall \, i \to j \in A, \quad R \leq x_{t \to s} \leq R$$

If quad or dual loads or stores have flexible memory layouts then this can be supported in the network flow formulation. For dual or quad load instructions the flow into the 2 or 4 variables being loaded is set to 1 or 2 respectively. For example in figure 5 the flow into the quad memory access variables is fixed to two. Mathematically we add the following constraint for dual or quad loads, $W \leq \Sigma_{k|k \in Q_i} \; \Sigma_{j|j \to v_k} \; x_{j \to v_k} \leq W, \; \forall i$ where $W=1$, for dual loads or $W=2$, for quad loads. Dual and quad stores are similar except the flow constraint is set on the output arcs leaving the dual/quad variable vertices (since the lifetimes of these variables end on the same clock cycle).

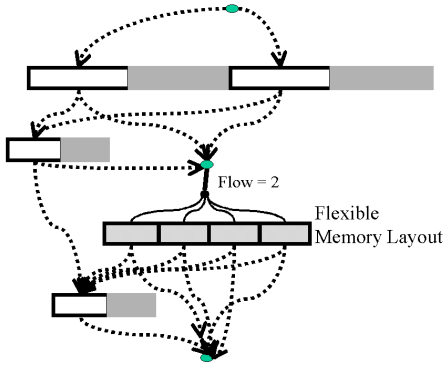

Figure 5. Flexible quad memory load with flow of 2.

In general more complex codes which have nested loops with dual and quad memory accesses are supported in this framework. In these cases the inner most nested loops are used to solve single network flow problems for each loop feedback variable. Multiword memory access instructions inside nested loops can be supported by early binding of the feedback variables to *d0,d1*, etc for each run of the network flow problem.

5. EXPERIMENTAL RESULTS

Several DSP applications are used to illustrate this methodology. The notation *dct, fir, red, pad* refers to a discrete cosine transform, linear FIR filter, a modulo reduction polynomial algorithm, and a polynomial addition algorithm (the last two examples used in cryptography) respectively. Assembly code was generated using the Star*core C compiler for all examples except the *fir* example where benchmark Motorola assembly code was used. The original C-compiler generated *dct* assembly codes contain 39 execution sets in the compiler generated assembly code. In *dct*, only single memory access instructions were utilized in the C compiler generated assembly code. The *fir* example has dual nested loops with high degrees of parallelism (execution sets of 7 words each cycle), using dual and single moves. The *red* example involved modulo reduction of 326 bit number in cryptography. Input data consisted of eleven 32 bit words allowing only dual long word loads to be supported. In the *pad* example, computations are performed on two 163 bit numbers within a single loop. Again in this example quad loads are not considered since they are only applicable in the Star*core

processor to 16 bit words. The maximum cost network flow problem was solved on a Sun workstation using a linear programming solver[8]. All optimizations ran in under 5 cpu seconds in total for each application on a 300MHz UltraSparcIIi Sun workstation.

Table 1 illustrates the optimized results (Opt-) compared to the C compiler generated code (Comp-). The number of registers *R*, the number of dual memory accesses D_1 and total memory access instructions, M_1, (in the original compiler generated code and) after the first implementation of maximum cost network flow is shown. The number of quad memory accesses Q_2, the number of dual memory accesses D_2 and the final number of memory access instructions, M_2, after second network flow application and complete register binding are also shown in the table. Results were generated automatically from assembly code using the maximum cost network flow formulation and using the memory layout and alignment for data input and data output specified by the compiler. To explore the relationship between increasing multiword memory accesses versus the total number of registers used, the network flow problems were solved with different amounts of flow (or number of registers). The multiword memory accesses can often be improved through an increase in the number of registers used. The improved memory bandwidth is shown as *%BW*, which we define as the number of memory access instructions in the original compiler-generated code divided by the number of memory access instructions in optimized code (after network flow optimization) shown as a percent.

The register binding for the cosine transform (*dct*) example with 12 registers maximum and three quad memory accesses involved one quad access of coefficients where flexible memory layout was allowed (like in figure 5) and two quad memory accesses of input data. The register binding methodology solved an initial network with flow of 4 (since the two quad memory accesses overlapped, similar to figure 4b)), two network flow problems each with flow one for each odd data register (as in figure 3b)) , one final odd data register network with flow of 2 and one last network flow for all remaining variables which was a flow of 4. Each flow problem was solved in under one second. The cosine transform was up to 1.76 (37/21) times more efficient in terms of memory accesses.

The *red* example used 32 bit word memory accesses so only dual memory accesses were supported (since quad accesses were only supported for 16bit words, using a 64 bit bus in [5]). In this example, up to 1.87 times improvement in memory accesses was observed. The benchmark *fir* assembly coded example had different memory accesses which reduced the overall code size by 1 word. This example had a loop which involved solving the network flow problem for interloop variables with single flows in addition to constraining any dual and quad memory access allocations. All results shown were verified and included loop support and register constraints on dual and quad memory accesses.

The *pad* example, in table 1, was also executed in hardware (using the Star*core chip on a development board) in order to measure the effect on power dissipation[17,18]. The memory bandwidth optimization technique provided one third reduction in energy dissipation (reduced from 4.8nJ to 3.0nJ). This energy dissipation was due to improved performance (15 cycles versus 26 cycles of compiler generated code) as a result of elimination of execution sets from allocation of dual loads and dual stores (and elimination of some address register load instructions) within the main loop of the *pad* example.

Table 1. Compiler versus Optimized Memory Accesses.

Ex	R	D_1	M_1	D_2	Q_2	M_2	*%BW*
Comp-*dct*	16	0	37	0	0	37	-
Opt-*dct*	12	9	28	5	3	23	60%
	13	10	27	3	4	22	68%
	14	11	26	1	5	21	76%
Comp-*fir*	12	6	10	6	0	10	-
Opt-*fir*	10	6	10	4	1	9	11%
Comp-*red*	12	0	15	0	0	15	-
Opt-*red*	9	6	9	6	-	9	66%
	10	7	8	7	-	8	87%
Comp-*pad*	6	0	11	0	0	11	-
Opt-*pad*	4	3	6	3	-	6	83%

6. DISCUSSION AND CONCLUSIONS

In summary code size overheads (including memory accesses) were improved up to 1.8 times using the maximum cost network flow formulation of the memory-access code problem. Unfortunately there is no previously published research to compare with, however results reported were compared with the Star*core's C compiler[13]. The network flow improved codes showed significant improvements. The technique presented in this paper performs register allocation and binding to minimize code size. The approach can be extended for loop support. Although several network flow problems are solved, each can be solved in polynomial time with faster network solvers. Furthermore only memory bandwidth utilization and it's impact on code size reduction was explored in this problem. It is possible that the improved memory bandwidth may also lead to improves in application throughput (through rescheduling). Although the paper has illustrated the network flow technique for 8 registers, it was implemented for 16 registers in the experimental section. For larger than 16 registers, a larger number of network flow problems would have to be solved. However in the worst case for *N* registers, *N* network flow problems would be solved, which includes one initial flow to find a lower bound on registers, two network flows to allocated dual and quad memory accesses and in the worst case *N* network flow problems for *N* data registers. For loop support this number would increase, however the network flow graphs would be of different sizes. For example one would solve the network flow on smaller graphs representing variables in innermost nested loops and merging the flows into a vertex in the graph at a higher level (outside of the nested loops) to complete the register binding. In all cases the register binding network flows did not increase the number of total registers being allocated.

The formulation of a new problem and a maximum cost flow approach to solving it has been presented in this paper. Unlike previous research, we have studied register allocation and

binding for multi-word memory access. Important memory bandwidth improvement and code size savings have been presented. Notably significant savings in energy dissipation was also achieved with this new approach. We have introduced a new methodology for utilizing memory bandwidth. It is applicable to other DSP processors (both fixed point and floating point ones) in addition to Star*core. For the first time, codesize-minimized memory-access code can be generated using this new application of the maximum cost network flow technique.

This research was supported in part by grants from Motorola (thanks to T.Tam, Y.Ronen, P.Marino, C.Hughes), NSERC and CITO.

7. REFERENCES

[1] P.Marwedel, G.Goossens Eds. *Code Generation for Embedded Processors*, Kluwer Academic Pub, 1995.

[2] B.Mesman,etal. "Constraint Analysis for DSP Code Generation", IEEE Transactions on CAD, Jan 1999.

[3] S.Wuytack, F.Catthoor, L.Nachtergaele, H.DeMan. "Min. the Required Memory bandwidth in VLSI sys", IEEE Transactions on VLSI, Dec1999.

[4] S.Liao, S.Devadas, K.Keutzer, S.Tjiang, A.Wang. "Storage Assignment to Decrease Code Size" PLDI, 1995.

[5] "Star*Core 140 Family DSP Core Instruction Set", Motorola/Lucent, Sept 1999.

[6] E.Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, RinehartWinston, 1976

[7] R.Leupers, P.Marwedel, "Time-Constrained Code Compilation for DSP's", IEEE Transactions on VLSI, 1997.

[8] A.Brooke, D.Kendrick, A.Meeraus, *GAMS, A User's Guide*, Scientific Press, 1988.

[9] G.Araujo, A.Sudarsanam, S.Malik. "Using RT Paths in Code Generation for Heterogenous Memory-Register Architectures" Proceedings of Design Automation Conference, 1996.

[10] J.Chang, M.Pedram "Register Allocation and Binding for Low Power" Proceedings of Design Automation Conference,, 1995.

[11] C.Liem, P.Paulin, A.Jerraya. "Address Calculation for Retargetable Compilation and Exploration of ISA" , Proceedings of Design Automation Conference, 1996.

[12] C.Gebotys " A Minimum Cost Circulation Approach to DSP Address-Code Generation", IEEE Transactions on CAD, Vol 18, No. 6, June 1999, pp726-741.

[13] "Star*Core 100 Family C/C++ Compilers User's Manual", Motorola and Lucent, Sept 1999.

[14] C.Gebotys, "Network Flow Approach to Data Regeneration for Low Energy Embedded Systems Synthesis", Integrated Computer-Aided Engineering, 5(2), 1998, p117-127.

[15] E.Lawler, Combinatorial Optimization: Networks and Matroids, New York: Holt,Rinehart and Winston, 1976.

[16] Nemhauser, Wolsey , Integer and Combinatorial Optimization, Wiley Interscience, New York, 1988.

[17] C.Gebotys, R.Gebotys, "Statistically based prediction of power dissipation for complex embedded DSP processors", Microprocessors and Microsystems, 23, pp135-144, 1999.

[18] C.Gebotys, R.Gebotys, S.Wiratunga "Power minimization Derived from Architectural-Usage of VLIW Processors", DAC, pp308-311, 2000.