Andersen's Points-to Analysis for CASH Compiler Framework

Deepak Garg Himanshu Jain

July 9, 2005

Project URL: http://www.cs.cmu.edu/∼hjain/compilers.html

Abstract

We present the design and results of our implementation of Andersen's pointer analysis on the Pegasus intermediate representation. We use a constraint based formulation of this analysis and use the Banshee constraint solving framework to solve the generated constraints. The results of the pointer analysis are then used for analyzing the token dependencies between the various memory related operations in the Pegasus IR. Our experimental results indicate that significant percentage of token dependencies in Pegasus IR are redundant. These dependencies can be removed to increase the parallelism in the Pegasus IR.

1 Introduction

Points-to analysis computes, for each expression in a given program, a set of possible abstract memory locations that the expression could point. Since the general problem of deciding points-to sets for each expression in a program is undecidable, any points-to analysis can produce approximate information only. There are several design choices that must be made while selecting a suitable pointer analysis for a given application. The main criteria used in making such a selection is a space-time vs accuracy trade-off. A *context sensitive analysis* tracks points-to sets for each call to a function separately. Such analyses are very costly in terms of time. A context-insensitive analysis, on the other hand, disregards such call information. Context insensitive analyses scale well in practice. In the rest of this report, we consider context insensitive analyses only.

Pointer analyses may also be classified as flow sensitive and flow insensitive. A flow sensitive analysis computes a points-to set for each pointer variable at each

program site. On the other hand, a flow insensitive analysis computes a single points-to set for each pointer variable. The latter form of analysis is less accurate, but runs reasonably well in practice. Two well-known flow insensitive points-to analyses are Steensgaard's analysis [8] and Andersen's analysis [1]. The latter is more accurate and can be implemented to run in worst case time $O(n^3)$, where n is the size of the program. In practice, however, it runs much faster. It is this analysis that we have chosen to implement for our application.

A very simple way to make a flow insensitive points-to analysis as accurate as a flow-sensitive one is to convert the program to SSA (static single assignment) form first. The SSA form of a program is an equivalent program in which each variable is assigned to exactly once and each use of a variable is preceded by its (unique) assignment on any program path reaching that use. Once a program has been converted to SSA form, a flow insensitive points-to analysis is as accurate as a flow sensitive one, because each variable is assigned exactly once. The difficulty however comes in obtaining the SSA form in the first place. As Hasti and Horwitz observe in [7], in order to obtain an SSA form for a program with pointers, at least a flow-insensitive pointer analysis must be performed first. Once this has been done and an SSA obtained, the same analysis can be performed again to obtain more accurate results.

The CASH framework is an experimental compiler that compiles C programs to asynchronous circuits [4]. Pointer analysis is important for CASH framework for two reasons. First, in order to obtain the circuit representation of a given C program, the program's dataflow graph must be converted to single static assignment (SSA) form first. As mentioned above, this requires at least a flow-insensitive analysis. The CASH compiler framework already has a flow-sensitive pointer analysis implemented for this purpose.

The second reason for implementing a pointer analysis in CASH stems from the design of its intermediate representation (called Pegasus [3]). In Pegasus, programs are represented as high-level circuits which contain functional nodes (like adders, logical operators, etc.), memory nodes (load and store) and a number of other flow related nodes like multiplexers, switches, etc. These nodes are connected by wires, and the entire intermediate representation takes the form of a graph where the nodes represent the program's operations and the connecting wires (edges) represent the dataflow. Since the circuit is asynchronous, control flow must be represented explicitly. Pegasus uses *token* edges to do this. If there is a token edge from node A to node B, then the function in node B can be performed only after the function at node A has computed its value. In most cases, token edges are needed between memory operations. For example, if a store node S writes to memory location *l* and a load node L has to read this value from location l, then node L cannot read l till node S has completed its write operation. In this case,

there would be a token edge from S to L.

During construction of the intermediate representation in CASH, a lot of token edges are added to ensure correct execution order. However, many of these token edges are redundant i.e. they connect load and store nodes which have no real dependency on each other. Such token edges constrain parallelism in the program. As a result it is desirable to remove them. This can be done using an accurate pointer analysis followed by simple dependence checks for memory operations that are connected by token edges. This pointer analysis can be implemented over the Pegasus representation itself, rather than the source program. This is a real advantage because the Pegasus representation is SSA (the variables in a Pegasus representation are the wires, and each wire is assigned exactly once – at its source) and hence a pointer analysis implemented on it is more accurate than an equivalent analysis on the source program.

In this report we present the design and results of our implementation of Andersen's analysis on the Pegasus intermediate representation. We use a constraint based formulation of this analysis and use the Banshee constraint solving framework [2] to solve constraints. The basic formulation of this analysis using Banshee as the constraint solver is not new. Details of a similar formulation can be found in [5, 6]. After performing the points-to analysis, we use it to locate token dependencies between load and store nodes that are redundant. Our experiments indicate that a significant percentage of dependencies in CASH's IR are redundant.

2 Andersen's Analysis on Pegasus IR

In this section we adapt an existing presentation of Andersen's analysis [5, 6] to the Pegasus IR. We skip a detailed description of the IR itself. This description may be found in [3]. Our points-to analysis is intra-procedural and is performed one function at a time.

We associate a name ($U, V, W, \ldots$) with each wire in the Pegasus representation of the function we are analyzing. With each wire name W we associate a symbolic name l_W, which is the name of the abstract memory location where the value present on the wire is stored[1]. To each wire W we associate a *type* τ_W which has the form $ref(l_W, X_W)$, where X_W is a *set* of types that the value in the wire may point to. We use the notation ϕ for the empty set and $\mathcal{U}$ for the set of all types. Formally, the types (τ) of wires are generated by the following grammar.

$$\begin{array}{lcl} \tau & ::= & ref(l_W, X_W) \\ X_W & ::= & \phi \mid \mathcal{U} \mid \{ref(l_{W_1}, X_{W_1}), \ldots, ref(l_{W_n}, X_{W_n})\} \end{array}$$

[1] In reality, some wires may actually get bound to hard registers, but for the purpose of our pointer analysis, we assume that all wires are allocated to memory locations

As a simple example, suppose we have a wire W which may point to wires V and U, which in turn may respectively point to wires of types X_V and X_U. Then the type of wire W is $\tau_W = ref(l_W, \{ref(l_V, X_V), ref(l_U, X_U)\})$. In order to obtain the exact set of locations a particular wire W may point to, we need to look at the set X_W in its type $ref(l_W, X_W)$. W may point to location l_V if and only if X_W contains a type of the form $ref(l_V, \_)$. Our points-to analysis implementation iterates over the entire circuit once and generates constraints between the sets X_W for different wires. Then we use the Banshee constraint solver to solve these constraints, thus obtaining the types of all wires in the circuit.

We generate constraints by iterating over the nodes (operations) of the Pegasus IR. For each operation in a function, constraints are generated between the points-to sets of the types of the input and output wires of the node. We describe in some detail how these constraints are generated.

Copy type nodes. Copy type nodes are those nodes which simply copy their input to the output. These include nodes having opcode `op_hold, op_reg, op_cast, op_nop, op_eta`. Suppose some such node N has input wire W and output wire V, with associated types $ref(l_W, X_W)$ and $ref(l_V, X_V)$ respectively. Then clearly, anything that W can point to, V can also point to, and as a result the constraint generated is $X_W \subseteq X_V$. Since the Banshee constraint solver computes least fixed points, in the absence of any more constraints, this implies that X_V will have the same value as X_W. This treatment also extends to nodes having opcode `op_neg`.

Load nodes. A load node (`op_lod`) has one input (the memory address from which to load) and one output, which is the value stored at that memory address. Let us call these input and output wires W and V respectively and let their corresponding types be $ref(l_W, X_W)$ and $ref(l_V, X_V)$. Symbolically, the node performs the operation $V = *W$. For every type $ref(l_U, X_U) \in X_W$, dereferencing W can result in a value of type X_U and hence, for each such U, we must have the constraint $X_U \subseteq X_V$. Banshee provides a convenient *projection* notation for writing such a constraint. We write $X_W \subseteq proj(ref, X_V, 1)$, which means that whenever $ref(l_U, X_U) \in X_W$, we have the constraint $X_U \subseteq X_V$.

Store nodes. A store node (`op_str`) has two inputs – a wire W which has the address at which the address must be stored and a wire V which holds the value that must be stored. There is no data output. Symbolically, the operation performed is $*W = V$. The constraint added for such a node is the following: for every location U that W points to, X_U must contain the set X_V. Formally, for each $ref(l_U, X_U) \in X_W$, we add the constraint $X_V \subseteq X_U$. Again, Banshee's

projection notation can be used to do this conveniently.

Constant nodes. If a node with opcode `op_const` is not a symbolic constant, then the output of the node cannot be a pointer. If the output wire of such a node is W, we add the constraint $\phi \subseteq X_W$. Since Banshee computes least fixed points, solving this system of constraints yields $X_W = \phi$. In words this means that the value on wire W is not a pointer.

If a node with opcode `op_const` is a symbolic constant 'c', we assume that we have an abstract location l_c in memory where 'c' resides. In this case, the output wire W is a pointer to the location l_c and the constraint we add is $\{ref(l_c, X_c)\} \subseteq X_W$, where X_c is a fresh set variable. If no further constraints are added on X_c, then X_c will evaluate to ϕ. In words this would mean that the wire W points to location l_c, which itself is not a pointer.

Logical operations. For operations, with opcodes `op_le, op_eq, op_land, op_lor, etc`, the output cannot be a pointer. If the output of such an operation is a wire W, we add the single constraint $\phi \subseteq X_W$.

Mathematical/bitwise/shift operations. These include nodes with opcodes `op_add, op_sub, op_and, op_or, etc`. For these operations, there are two inputs (say U and V) and one output (say W). In these cases, we simply assume that W can point to anything that U or V can point to i.e. $X_U \subseteq X_W$ and $X_V \subseteq X_W$. In the most common case, such a node is used for pointer arithmetic operations, and one of the inputs (say V) is a constant. For example, one may increment the value of a pointer by a constant value 4 to point to the next element of an integer array. In such cases, the value X_V will be ϕ and the only useful constraint is $X_U \subseteq X_W$. As a result, our pointer analysis identifies the different fields of an array or a structure in its points-to sets.

Mu/Mux/Switch nodes. In these cases, there are a number of input wires (say $W_1, \ldots, W_n$) and a single output wire (say W). The value on exactly one of the wire $W_1, \ldots, W_n$) is copied to the output. Therefore we add n constraints – for each $1 \leq i \leq n$, we add the constraint $X_{W_i} \subseteq X_W$.

Arguments to functions. Nodes with opcode `op_arg` require special treatment because their output can potentially point to any location other than those created in the function body. We use the special location name l_{arg} to denote all such locations. Correspondingly, we use the name X_{arg} to denote the set of all possible types that can be obtained by dereferencing l_{arg}. We assume that each argument to a function is a pointer to this location. As a result, if W is a wire at the output from

an argument node, we add the constraint $\{ref(l_{arg}, X_{arg})\} \subseteq X_W$.

Function calls. If a function call is made to an address with symbolic name "malloc", then the output is a new location. In this case, if the output wire is W, we add the constraint $\{ref(l_f, X_f)\} \subseteq X_W$, where l_f and X_f are fresh names. If the call is to some other function, the return value can potentially point to anything and hence the constraint added is $\mathcal{U} \subseteq X_W$.

Once constraints have been generated for all nodes in a function's IR, we can solve them using Banshee. However, for purposes of efficiency, we solve the constraints on a need-only basis. When we are looking for token dependencies between nodes, we do not need to solve all constraints. Instead we need to solve only those constraints which affect the inputs and outputs of the specific nodes we are considering. Therefore, we postpone solving the constraints to the time when their values are actually needed.

3 Identifying token dependencies

False token dependencies between operations reduce the parallelism available in a circuit. In this section we describe how the points-to information can be used for identifying false token dependencies between operations. This in turn can be used for removing certain token edges from the circuit, thus, leading to increased parallelism.

A token edge indicates a dependence between the connecting nodes (operations). However, there might be a dependence between two nodes even if they are not directly connected by a token edge. In general, there is a *token dependency* from node n to node m if there is a path consisting of token edges from node n to node m. We identify the token dependencies by computing the transitive closure of the graph induced by the token edges. Once we have obtained all the token dependencies, we classify them as false (i.e., can be removed) or true (i.e., cannot be removed). This classification process is described next.

For each node n, let $read(n)$ denote the set of abstract memory locations read by the node n, and $write(n)$ denote the set of abstract memory locations written by the node n. The computation of read and write sets makes use of the points-to information. For example, for a load node l of the form `x = *y`, we have $read(l) = \{y, pts(y)\}$, where $pts(y)$ denotes the set of abstract memory locations pointed to by y. Similarly, $write(l) = pts(x)$.

A token dependency between two nodes n and m is false iff the following

equation holds:

$$((read(n) \cap write(m) \ = \ \emptyset) \wedge (write(n) \cap read(m) \ = \ \emptyset)) \wedge \\ (write(n) \cap write(m) \ = \ \emptyset))$$

The above equation is expressed using the set constraints of Banshee. The constraints corresponding to the above equation are then solved to determine if there is a false token dependency between the nodes n and m.

Since most of the token dependencies arise due to memory operations, we perform the above computation (transitive closure of token edges, constraint solving) only for the load/store nodes. This is crucial because performing transitive closure for all the nodes in the circuit is expensive and intractable for large programs.

4 Experimental results

The experimental results are summarized in Table 1. The column "Benchmark" contains the benchmark name, the column "TP" contains the total number of procedures analyzed in a given benchmark. The "Nodes" column contains two sub-columns: sub-column "TN" gives the total number of nodes in the various Pegasus circuits of a given benchmark; sub-column "L/S" is the total number of load or store nodes in these circuits. The constraint generation is done for all the nodes in a circuit. However, the constraint solving is done only for load/store nodes which have token dependencies between them.

The column "Dependencies" refers to the token dependencies between load/store nodes: sub-column "TrueD" gives the total number of token dependencies which are true; sub-column "FalseD" gives the total number of token dependencies which are false.

The column "Andersen's Time" contains the time taken by various steps of our implementation of Andersen's analysis. The sub-column "CGen" gives the total time taken by constraint generation step; the sub-column "Tk" gives the total time taken when computing the transitive closure of load/store nodes; the sub-column "Dep" gives the total taken for analyzing the token dependencies between load/store nodes; the sub-column "Tot" gives the total time taken by our analysis. The time taken by existing pointer analysis (done at source level) is given in the column "ET".

4.1 Summary of Results

- The total number of false token dependencies is greater than the total number of true dependencies for the majority of benchmarks. Thus, many token

edges can be removed from the circuit. However, computing the set of token edges that can be removed is not straightforward. This is because a given token edge can participate in multiple token dependencies, some of which can be true and cannot be removed. One way to get around this problem is to first remove the entire token network and then re-build it based on true token dependencies only.

- We observed that computing the transitive closue of token edges for all the nodes is too expensive. Thus, we only compute the set of nodes reachable from each load/store node. This speeds up the analysis as the total number of load/store nodes is smaller as compared to the total number of nodes (column "Nodes").

5 Conclusion

We have presented a constraint based formulation of the Andersen's points-to analysis for the Pegasus intermediate representation. We described the constraint generation rules for various operations in the Pegasus IR. The Banshee toolkit is used for solving the generated constraints. The results of our points-to analysis are used to identify the redundant token dependencies between load and store nodes. Our experiments indicate that a significant percentage of token dependencies in CASH's IR are redundant.

Pegasus IR is in SSA form, and thus, even a flow-insensitive analysis such as Andersen's produces the same results as a flow sensitive analysis. However, one drawback of the Pegasus IR is the lack of source level type information. Without the type information we do not know how many times a pointer variable can be dereferenced (e.g., `int**` can be dereferenced twice). This sometimes leads to generation of constraints which cannot be solved by Banshee. Other limitations are: 1) we do not track function pointers because our analysis is intra-procedural. 2) we cannot detect aliasing between function arguments and global variables because we cannot detect which symbol names in a circuit are global.

References

[1] L. Andersen. Program Analysis and Specification for the C Programming Language. Technical Report Ph.D. thesis, DIKU, University of Copenhagen, 1994.

[2] http://banshee.sourceforge.net/.

[3] M. Budiu and S. C. Goldstein. Pegasus: An Efficient Intermediate Representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, 2002.

[4] http://www-2.cs.cmu.edu/∼phoenix/index.html.

[5] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Flow-Insensitive Points-to Analysis with Term and Set Constraints. Technical Report CSD-97-964, University of California, Berkeley, 1997.

[6] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *Static Analysis Symposium*, pages 175–198, 2000.

[7] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 97–105, New York, NY, USA, 1998. ACM Press.

[8] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 32–41, 1995.

Bench-mark	TP	Nodes		Dependencies		Andersen's Time				ET
		TN	L/S	TrueD	FalseD	CGen	Tk	Dep	Tot	
099.go	210	267568	6388	15486	185785	3.4	10.56	15.58	29.98	3.06
129.compress	15	7800	324	5273	3514	0.14	0.1	0.57	0.82	0.03
130.li	72	18168	831	1945	16756	0.27	0.34	0.98	1.59	0.17
132.ijpeg	13	10784	566	7572	16995	0.13	0.39	1.46	2.02	0.05
147.vortex	72	95440	2791	6631	105670	0.98	5.34	6.04	12.48	14.47
164.gzip	14	17792	524	2332	23003	0.25	0.99	1.4	2.66	0.17
164.gzip.test	14	17856	516	2305	22142	0.21	1.1	1.44	2.76	0.16
175.vpr	29	26856	844	409	26022	0.32	0.89	1.48	2.74	0.5
176.gcc	40	10992	421	376	6379	0.12	0.32	0.42	0.88	0.1
179.art	3	984	46	80	549	0.02	0.01	0.03	0.07	0
181.mcf	23	21976	1093	30423	34858	0.29	0.57	6.16	7.04	0.14
183.equake	3	8880	262	308	35662	0.09	1.11	1.79	3.01	0.35
188.ammp	3	2304	65	112	688	0.04	0.02	0.06	0.12	0.04
197.parser	83	57216	1759	12100	71193	0.69	2.9	5.73	9.38	0.73
253.perlbmk	22	13776	780	2207	18462	0.17	0.3	1.31	1.8	0.25
254.gap	34	29312	1375	2583	78481	0.39	0.98	4.88	6.29	0.27
255.vortex	72	95440	2791	6631	105670	0.99	5.59	6.37	13.16	15.22
256.bzip2	62	78376	2226	22567	257037	1.08	12.76	18	31.96	0.94
300.twolf	15	28944	1551	25886	132042	0.34	1.64	9.5	11.48	0.32
301.apsi	1	16688	154	1223	9204	0.16	3.84	0.54	4.56	60.4
adpcm_d	1	1192	19	36	104	0.01	0.01	0.01	0.04	0.01
adpcm_e	1	2248	20	32	82	0.02	0.04	0	0.08	0.02
basicmath	4	8784	77	10	1269	0.09	0.35	0.05	0.51	0.09
blowfish_d	3	4920	330	4625	6740	0.11	0.11	0.67	0.9	0.03
blowfish_e	3	4920	330	4625	6740	0.11	0.11	0.66	0.89	0.03
fir_wagner	1	648	31	175	45	0.01	0	0.02	0.03	0.01
g721_d	20	5752	160	1538	262	0.06	0.06	0.11	0.24	0.04
g721_e	15	4496	151	1531	213	0.03	0.03	0.11	0.18	0.01
g721_Q_d	20	6008	157	1322	218	0.05	0.04	0.09	0.19	0.02
g721_Q_e	15	4752	148	1315	169	0.08	0.04	0.08	0.2	0.03
g721_Q_e_custom	15	4752	148	1315	169	0.05	0.05	0.08	0.19	0.02
gsm_d	79	56920	1831	76202	53249	0.54	1.74	7.95	10.32	0.49
gsm_e	80	57392	1844	75935	53163	0.56	1.66	8.16	10.43	0.55
ispell	106	102032	2306	10502	77958	1.14	5.48	5.32	12.09	2.66
jpeg_d	13	11640	482	7140	15944	0.21	0.41	1.31	1.96	1.08
jpeg_e	20	14824	669	6859	18162	0.19	0.47	1.44	2.17	1.26
mi_jpeg_d	13	11640	482	7140	15944	0.22	0.42	1.34	1.99	1.17
mi_jpeg_e	20	14824	669	6859	18162	0.16	0.48	1.49	2.17	1.27
mpeg2_d	20	12288	529	459	35897	0.15	0.34	1.94	2.44	0.16
mpeg2_e	2	8096	118	0	3884	0.11	0.34	0.19	0.64	0.25
pegwit_d	58	29128	706	5706	9070	0.35	0.44	0.99	1.84	0.2
pegwit_e	58	29128	706	5706	9070	0.35	0.4	0.98	1.75	0.2
pgp_d	104	72704	1116	5302	12016	0.72	1.76	0.94	3.48	2.48
pgp_e	104	72704	1116	5302	12016	0.67	1.68	1	3.48	2.4
qsort	3	3464	98	4690	2450	0.04	0.2	0.62	0.86	0.04
rasta	3	656	18	12	57	0.01	0	0	0.02	0.01
sha	5	696	21	10	0	0	0	0	0	0.02
sumofsquares	2	336	0	0	0	0.01	0	0	0.01	0
susan	13	21864	652	8314	66531	0.32	0.86	14.99	16.2	0.1

Table 1: Andersen's pointer analysis results