2007－EVA－20 (5)
2007／3／16

ハードウェア・ソフトウェア協調エミュレーション・シミュレーション手法

木下 修平† 並木 滋† 近藤 信行† 中島 隆仁‡ 清水 尚彦††
東海大学大学院 工学研究科†
東海大学 電子情報学部 コミュニケーション工学科‡
東海大学 情報理工学部 ソフトウェア開発工学科††

近年、SoC(System on chip) やカスタム LSI の開発は急速に大規模化・複雑化が進み、それに伴う開発時の機能検証時間の増加が問題視されている。こうした中で、検証効率の向上を目的として協調検証環境が提案されている。協調検証の一種である協調エミュレーションは、検証環境論理をハードウェア上に載せ、それらをソフトウェアから制御することで検証を行う方法である。我々はハードウェアとして PCI バス上に搭載した FPGA を用いた協調検証環境を開発し、その評価を行った。

Tecnique of Co-Emulation and Co-simulation

Shuhei Kinoshita†, Shigeru Namiki†, Nobuyuki Kondoh†,
Takahito Nakajima‡ and Naohiko Shimizu††
Graduated School of Engineering, Tokai University†
School of Information Technology and Electronics, Tokai University‡
School of Information Science and Technology, Tokai University††

In recent years, many very highly efficient and highly efficient large-scale integrated circuits called a system LSI are developed. And the verification in those LSI developments takes time very much. In such the situation, Co-verification environment was proposed to solve that problem. Co-Emulation is a kind of Co-verification method. It verify the hardware logic using software that control the verification environment hardware. We developed the Co-Verification environment which used FPGA and PCI Bus, and evaluated it.

1 はじめに

近年、SoC(System on chip) やカスタム LSI の開発は急速に大規模化・複雑化が進み、それに伴う開発時の機能検証時間の増加が問題視されている。そのような中で、検証時間の短縮を目的として協調検証という方法が提案されている。協調検証環境は機能検証を行うためのソフトウェアコード (テストベンチ) から被検証ハードウェア論理 (DUT:Design Under Test) を検証する検証方法である。ハードウェアのみで論理検証を行う場合と比較して、テストベンチをソフトウェアで記述できることから制御が容易であるとともに、入出力データの観測性も高い。また、協調検証環境は協調シミュレーションと協調エミュレーションの２種類に大別される。協調シミュレーションは DUT を機能的に等価なソフトウェアモデル化し、テストベンチと共にソフトウェア上にて動作させることで検証を行う方法である。一方、協調エミュレーションは、検証対象論理を検証環境用論理と共にハードウェア上に載せ、ソフトウェアからデータ入出力及びクロック信号を制御することで検証を行う検証方法である。本稿では我々が開発した独自協調検証環境とその評価方法及び評価結果について報告する。

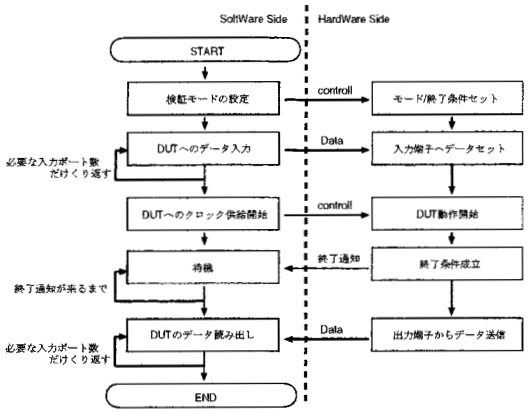


図 1: テストベンチの基本 1 ステップフロー

2 開発した協調検証環境

今回我々はハードウェアとして PCI バス上に搭載した FPGA を用いる協調エミュレーションと、Verilog シミュレータである Verilator[1] を用いる協調シミュレーションを同一のテストベンチから利用できる協調検証環境を開発した。協調シミュレーションと協調エミュレーションの大きな違いはその検証速度であり、一般的に各手法の検証速度には以下の特徴がある。

- 協調シミュレーション

  検証速度は検証対象の論理規模、シミュレーションを実行させる計算機の性能に依存する

- 協調エミュレーション

  検証速度はハードウェアーソフトウェア間の物理インタフェースのデータ伝送速度・検証環境ハードウェアの動作周波数に依存

小規模な論理や短時間の検証においては、協調シミュレーションのほうが検証速度が高速である場合もあるが、ある程度規模の大きい論理の検証においては協調エミュレーションのほうが高速な検証が行えることが多い。このため状況に応じて協調エミュレーション・協調シミュレーションを切り替えられることが望ましいと考え、2 手法に対応した検証環境を開発することとした。

開発した環境は以下のステップを繰り返すことで検証を行う。

1. 検証モードの設定
2. DUT 入力端子へのデータ入力
3. DUT へのクロック供給開始する (テストベンチは制御が戻ってくるまで待機状態)
4. DUT 出力端子のデータ取得

図 1 に 1 ステップの処理フローを示す。開発した環境は検証モードとしてサイクルモードとトランザクションモードの 2 モードをもつ。これは DUT のクロック信号制御をある定数単位で行うモードと、トランザクション (ユーザ定義の 1 処理) 単位で行うモードである。検証開始時は DUT へクロック信号は供給されていない。ユーザは DUT の入力端子に入力データをセットした後、DUT へクロック信号供給を開始することで検証を行う。クロック信号の供給後は、検証環境ハードウェア側 (DUT 側) からの終了通知が来るまでテストベンチ側は待機状態となり、各検証モードの条件を満たすことで終了通知がハード側から発行さ

れると再び動作を開始する。動作開始後、DUT の出力端子のデータを読み出すことで 1 ステップ完了となる。また、検証環境用の検証用ハードウェア及びテストベンチが用いる API などは DUT ソースコードから InfrastructureLinker と呼ばれるプログラムにより自動生成される。

3 性能評価

開発した環境において協調エミュレーションを行う際の性能評価を行った。性能評価は基本性能の評価を目的としており、協調エミュレーション時におけるデータの送受信性能を測定することとした。測定時に際しては、API 内の各処理を細分化しそれぞれの処理時間を計測すること検証動作全体におけるボトルネック部分を抽出する。処理時間の測定は、各処理に要したクロック数を rdtsc 命令を用いて測定し、経過クロック数を評価環境における CPU の動作周波数で除算することで行う。この際、デバイスに対する直接的なデータ送受信処理の測定部には mfence 命令を挿入することでアウトオブオーダなどの影響を抑え、より正確な送受信時間を求める。評価に用いた環境を表 3 に示す。

表 1: 評価環境

マシン名	DELL PowerEdge SC430
CPU	Celeron 2.53GHz
OS	debian Linux kernel 2.6.15.7
C++コンパイラ	gcc 4.1.2
FPGA ボード	Stratix EP1S10F780C7ES

3.1 ドライバの送受信用関数を用いたデータ送信

ドライバを利用したデータの送信時の経過時間の測定を行った。データ送信の処理フローを図 2 に示す。ユーザはテストベンチから API の Send() メソッドを呼ぶことでデータの送信を行う。データ送信処理は API 内処理とドライバ内処理の 2 処理に大別され、API 内処理においては WriteMessage 関数とその他の処理に分けることが出来る。さらに WriteMessage 関数内は

- 1. トランザクタ番号セット (ioctl(TRSCT_SET))
- 2. マッピングアドレスセット (ioctl(BIT_OFFSET_ADRS_SET)
- 3. データライト (write())
- 4. その他の処理

という処理に分割し各部の計測を行った。計測は 32bit の入力ポート 1 つに対して 10000 回送信を行い、その平均値を求めることで 1 回あたりの送信に必要な時間を求めた。計測結果を元に各処理の割合を算出したグラフを図 3 に示す。計測結果から、ドライバの関数呼び出しに必要な時間 (SystemCall) が総時間の 7 割程度を占める結果となった。このことから SystemCall 数を減らすことで大幅な性能向上が期待できることが判明した。

同じくドライバを利用したデータの受信時の経過時間の測定を行った。データ送信の処理フローを図 4 に示す。ユーザはテストベンチから API の ServiceLoop() メソッドを呼ぶことでデータの受信を行う。API においてデータの受信はユーザが出力ポートを指定して明示的に行うことは許されておらず、ServiceLoop() メソッドにより DUT へのクロック供給が行われた後で自動的にデータ受信処理が開始される。このため、

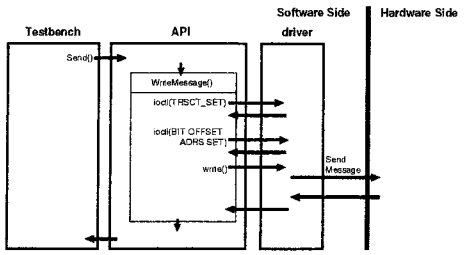


図 2: ドライバ送信関数利用時のデータ送信処理フロー

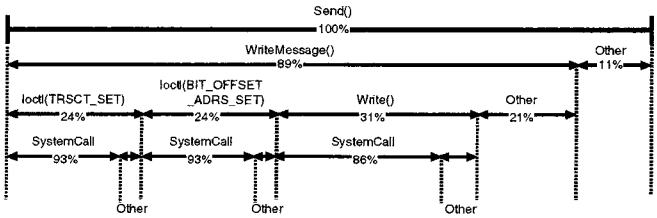


図 3: 32bit データ送信時の経過時間分析 (ドライバ送信関数利用時)

ServiceLoop() メソッドの処理を分別した場合、DUT クロック処理とデータ受信処理という 2 処理に分けることができる。データ受信処理はデータ送信処理と同様に API 内処理とドライバ内処理の 2 処理に大別され、API 内では ReadMessage 関数とその他の処理に分けることが出来る。そこで ReadMessage 関数内を

- 1. トランザクタ番号セット (ioctl(TRSCT_SET))
- 2. マッピングアドレスセット (ioctl(BIT_OFFSET_ADRS_SET)
- 3. データリード (read())
- 4. その他の処理

という 4 処理に分割し各部の計測を行った。計測は 32bit の出力ポート 1 ポートのデータ受信を 10000 回行い、その平均値を求めることで 1 回あたりの受信時間とした。計測結果を元に各処理の割合を算出したグラフを図 5 に示す。計測結果から、ドライバへの SystemCall が総時間の 5 割程度を占める結果となった。このことからデータ送信同様、SystemCall 数を減らすことで大幅な性能向上が期待できるといえる。

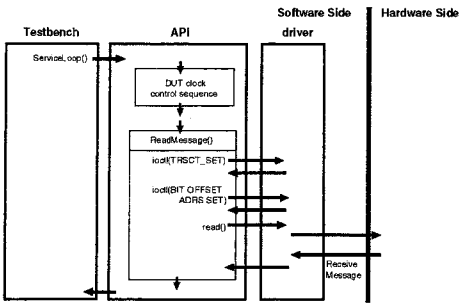


図 4: ドライバ受信関数利用時のデータ受信処理フロー

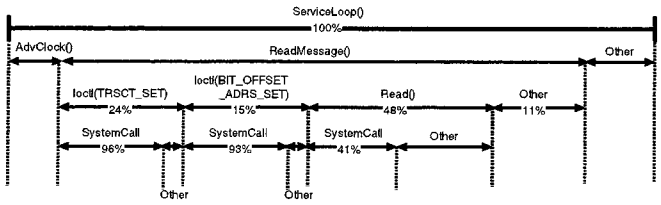


図 5: 32bit データ受信時の経過時間割合 (ドライバ受信関数利用時)

3.2 mmap を用いたデータ送受信

前項の性能評価結果から、SystemCall 削減による性能向上が期待できることが判明したため、mmap を用いて API から直接ハードウェア側へのデータ送受信を行うことができるよう改良を行った。mmap を用いることにより、ドライバがもつ PCI デバイスのメモリマップされたアドレスをユーザ空間のアドレス空間に割り付けることができる。このためデータ送受信時にドライバの write(),read() などの SystemCall を使わずとも、割り付けられたアドレスに対してデータを読み書きするだけで、デバイスに対するアクセスが可能となる。また、トランザクタ番号やマップアドレスをユーザ空間内で保持・処理すればよいため ioctl(TRSCT_SET) 及び ioctl(BIT_OFFSET_ADRS_SET) などを呼ぶ必要が無くなる。mmap を利用した際のデータ送信フローとその処理内訳をそれぞれ図 6, 図 7 に示す。ドライバ利用時と同様に計測は 32bit の入力ポート 1 ポートのデータ送信を 10000 回繰り返し行い、その平均値を求めた。計測結果を元に各処理の割合を算出したグラフを図 7 に、mmap を用いたデータ送信とドライバを用いたデータ送信の経過時間比較を図 8 に示す。mmap 利用時はドライバ利用時とくらべ、4 倍ほど時間短縮されていることがわかる。また、送信回数が増すにつれその差は拡大する。

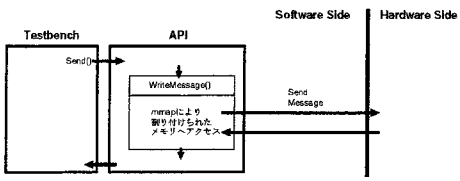


図 6: mmap 利用時のデータ送信処理フロー

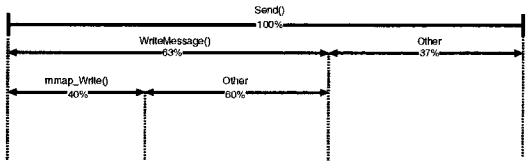
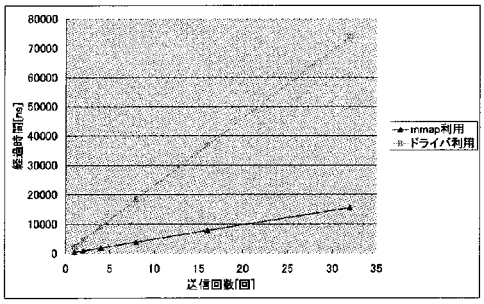


図 7: 32bit データ送信時の経過時間割合 (mmap 利用時)



送信回数	経過時間	
[回]	mmap[ns]	ドライバ [ns]
1	467	2284
2	940	4597
4	1930	9207
8	3928	18466
16	7862	36887
32	15574	73861

図 8: データ送信時間比較 (mmap 利用時 - ドライバ利用時)

mmap を利用した際のデータ受信フローと処理内訳をそれぞれ図 9, 図 10 に示す。受信についても計測は 32bit の出力ポート 1 ポートのデータ受信を 10000 回繰り返し行い、その平均値を求めた。計測結果を元に各処理の割合を算出したグラフを図 10 に、mmap を用いたデータ送信とドライバを用いたデータ送信の経過時間比較を図 11 に示す。mmap 利用時はドライバ利用時とくらべ、3 倍ほど時間短縮されていることがわかる。また、送信回数が増すにつれその差は拡大する。

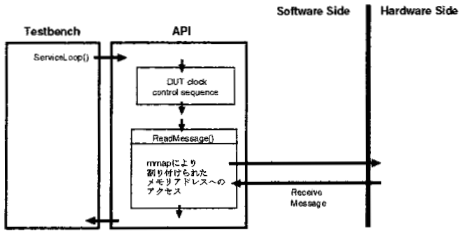


図 9: mmap 利用時のデータ受信処理フロー

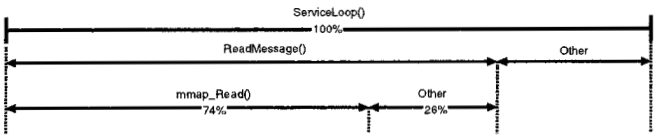
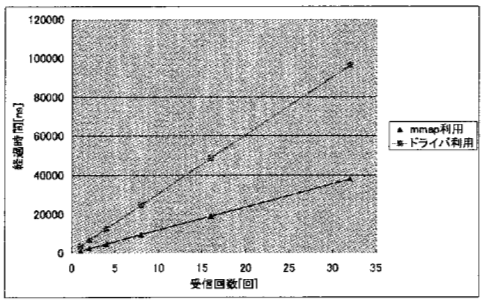


図 10: 32bit データ受信時の経過時間割合 (mmap 利用時)



送信回数	経過時間	
[回]	mmap[ns]	ドライバ [ns]
1	1160	3618
2	2391	6766
4	4677	12557
8	9547	24573
16	18883	48743
32	37921	96493

図 11: データ受信時間比較 (mmap 利用時 - ドライバ利用時)

3.3 データ送受信処理時間の詳細分析

前項において SystemCall 時間が無くなったことにより、API 内処理をより細分化することでボトルネックの検出を行うことが可能となる。そこでデータ送受信処理を細分化し、各部の経過時間の測定を行った。データ送信時、データ受信時における細分化した処理項目を図 12 に示す。

API から DUT の入出力ポートにアクセスする際にはドライバにより割り付けられたある一定の連続したメモリ空間にアクセスすることで行う。各入出力ポートは、先頭アドレスから隙間無く詰められた状態でマップされている。しかしながら設計した SCE-MI H/W ではデータ転送にかかる時間的コストを考慮し、32bit 単位のデータの読み書きしか行わない仕様となっている (32bit 単位のことを以降スロットと呼ぶ)。こ

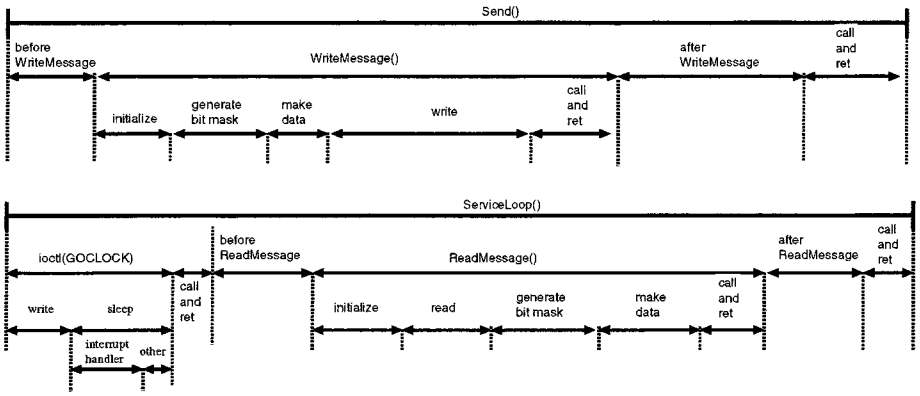


図 12: データ送受信時の処理項目

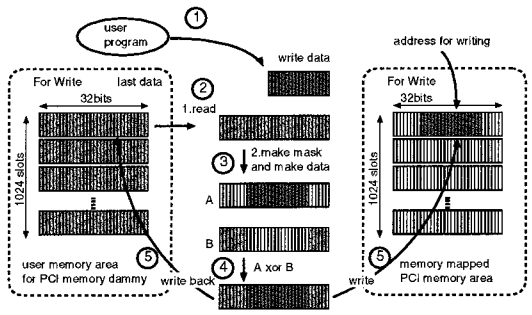


図 13: データ送信時のデータ処理例

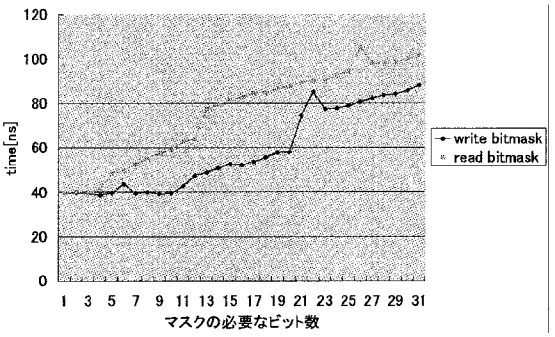


図 14: 単位スロット当りのマスク数とマスク処理時間

のため、各ポート用のマップアドレス・ポート幅などにより読み書きするデータの処理が異なる。つまり、ポートのマップアドレスが 32bit アライン上に存在し、かつポート幅が 32bit の整数倍の場合以外はデータ整形を行わないと、他のポートのデータを破壊する恐れがある。データ整形はビットマスクを作成し、それを用いて必要な部分のデータのみを読み書きする。図 13 に入力ポートのマップアドレスが 32bit アライン上に無い場合のデータ送信時のデータ処理例を示す。結果として、データ送受信を行う場合、データ整形を行う場合と行わない場合に処理を分けて考えることができる。そこでそれぞれの条件において計測を行った。

マスク処理を行わない場合の、データ送受信処理の計測結果を表 2 に示す。計測時間は 10000 回繰り返し処理を行った平均を求めた。結果より、データ送信時 1 スロットの書き込みに対して約 230[ns]、データ受信時 1 スロットの読み込みに対して約 911[ns] の時間が必要となることがわかる。開発した環境の PCI コントローラは送信/受信プロトコル処理にともに 4[clk] かかる仕様となっており、PCI バスの動作周波数は 33[MHz] であることからデータ送受信時にプロトコルが必要とする時間は 121[ns] であるといえる。このことと計測結果から、データ送受信時には PCI バスプロトコル以外の処理時間が割合的に大きなものとなっていることがわかる。この"プロトコル以外の処理"の詳細な処理内容と処理時間の計測方法については現時点では検討している段階である。次にマスク生成を必要とする場合について計測を行った。計測ではマップアドレスは 0 固定とし、ポート幅を 1〜31 まで 1 づつ変化させた際のデータ送受信時間を測定した。結果を図 14 と表 3 に示す。

表 2: 送受信の処理時間 (マスク処理なし)

スロット数	経過時間 [ns]	
	データライト	データリード
1	352	1035
5	1268	4700
10	2424	9307
15	3546	13842
20	4696	18483
25	5842	22920
30	7022	27457

表 3: 送受信処理時間 (マスク処理あり)(スロット数 1)

	経過時間 [ns]			
maskbit 数	write マスク処理	WriteMessage	read マスク処理	ReadMessage
0	0	351.89	0	1034.95
1	39.76	387.77	39.82	1076.10
5	39.66	388.109	48.72	1084.63
10	39.50	388.32	59.00	1096.14
15	52.51	402.65	81.68	1114.75
20	57.88	414.89	87.52	1123.87
25	78.93	431.41	94.20	1124.59
30	85.61	436.85	99.97	1133.87

結果よりマスクが必要となる状況かつ同一のスロットに対するデータ送受信時間はマスク処理時間に依存し、マスク処理時間はマスクビット幅に比例することがわかる。ビットマスクはプログラム内において、送受信のたびに for 文を用いて作成しているため、このような結果となったと考えられる。改善策の一例としては各ポートに対するマスクを InfrastructureLinker であらかじめ作成しておき、送受信時にはそのマスクを利用するように処理を変更するなどがあげられる。この改善策を用いるとマスク処理にかかる時間を削減することができるため、即座に性能向上に繋がると考える。

4 まとめと今後の予定

開発した協調検証環境の性能評価を行った。検証においてはプログラム内の処理を細分化して考え、各部の処理時間を測定することでボトルネックとなる箇所を抽出した。初期の段階におけるボトルネックであったデータ送受信時のシステムコール処理時間を mmap を用いることで減少させ、検証速度の向上を確認した。今後はより詳細な性能評価のための測定方法などについて検討を行うとともに、測定データを元に検証時間の概算式を求める。

参考文献

[1] Verilator
http://www.veripool.com/verilator.html

[2] 東海大学大学院工学研究科, 近藤 信行, 修士論文
「LSI 開発のための FPGA を用いた高速検証環境の構築」(2007)

[3] IP ARCH, Inc.
http://www.ip-arch.jp/

[4] Accellera, "SCE-MI Reference Manual DRAFT"

[5] 早坂, 志水, 横山, 孕石,"第 9 回 ASIC デザインコンテスト 規定課題 B PCI バスインタフェース", 第 22 回パルテノン研究会,2003

[6] J.Corbet,A.Rubini,G.Kroah-Hartman,"Linux Device Drivers",2005 Oreilly & Associates Inc

[7] TECHI PCI デバイス設計入門 PCI バスの原理から HDL による IC 設計&デバッグ手法まで", CQ 出版社